University of Crete
School of Sciences and Engineering
Computer Science Department

Practical Information Flow for Legacy Web Applications

Georgios Chinis

Masters' thesis

Heraklion, October 2012

University of Crete
School of Sciences and Engineering
Computer Science Department

Practical Information Flow for Legacy Web Applications

Thesis submitted by
Georgios Chinis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author:

Georgios Chinis

Committee approvals:

Evangelos Markatos
Professor, Thesis Advisor

Sotiris Ioannidis
Principal Researcher, Committee Member

Maria Papadopouli
Associate Professor, Committee Member

Departmental approval:

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, October 2012

Abstract

The popularity of web applications, coupled with the data they operate on, makes them prime targets for miscreants that want to misuse them. To make matters worse, a lot of these applications, have not been implemented with security in mind, while refactoring an existing, large web application to implement a security or privacy policy is prohibitively difficult.

This thesis presents LabelFlow, an extension of PHP that simplifies implementation of security policies in web applications. To enforce a policy, LabelFlow tracks the propagation of information throughout the application, transparently and efficiently, both in the PHP runtime and through persistent storage. We provide strong theoretical guarantees for the policy enforcement in LabelFlow; we define its semantics for a simple calculus and prove that it protects against information leaks. LabelFlow is applicable to real-world large scale web applications. We used LabelFlow to add and enforce access control policies in three popular web application MediaWiki, WordPress and OpenCart with small execution overhead and code changes.

Περίληψη

Η δημοφιλία των Web εφαρμογών, σε συνδυασμό με τα δεδομένα που διαχειρίζονται τις καθιστά πρωταρχικούς στόχος για κακόβουλους χρήστες που θέλουν να τις εκμεταλλευτούν. Η κατάσταση επιδεινώνεται από το γεγονός οτι πολλές εφαρμογες δεν έχουν σχεδιαστει με γνώμονα την ασφάλεια, ενώ η τροποποιηση μιας υπάρχουσας, μεγάλης Web εφαρμογής για να εφαρμόζει μια πολιτική ασφάλειας, είναι απαγορευτικά δύσκολη.

Σε αυτήν την εργασία παρουσιάζουμε το LabelFlow, μια επέκταση της PHP, που απλοποιεί την εφαρμογή των πολιτικών ασφάλειας σε εφαρμογές Web. Για την επιβολή της πολιτικής, το Λα- βελΦλοω παρακολουθεί τη ροη της πληροφορίας σε όλη την εφαρμογή, τόσο κατά την εκτέλεση του κώδικα όσο και στο χώρο μόνιμης αποθήκευσης. Παρέχου- με ισχυρές θεωρητικές εγγυήσεις για την εφαρμογή της πολιτικής. Ορίζουμε τη σημασιολογία των αλλαγών μας σε Lambda calculus και αποδεικνύουμε ότι το LabelFlow προστατεύει από διαρροές πληροφορίων. Χρησιμοποιήσαμε το LabelFlow σε τρείς πραγματικές εφαρμογές, MediaWiki, WordPress και OpenCart ώστε να επιβάλουμε πολιτικές ελέγχου πρόσβασης. Οι αλλαγές που χρειάστηκαν στον κώδικα ήταν ελάχιστες.

Επόπτης: Καθ. Ευάγγελος Μαρκάτος

Acknowledgements

I would like to thank my supervison, Professor Evangelos Markatos, for his valuable guideline in my academics steps in the field of Computer Science. I also feel grateful to Dr. Sotiris Ioannidis, who gave me the opportunity to work on this subject and provided me valuable guidelines and advices. Many thanks to Dr. Polyvios Pratikakis and Dr. Elias Athanasopoulos for their valuable help and contribution on my thesis. My best thanks to my friends and colleagues Lazaros Koromilas, Stamatis Volatis, Zaharias Tzermias, Thanasis Petsas, Giorgos Vasiliadis, Antonis Papadogiannakis, Iasonas Polakis, Christos Papachristos, Manolis Stamatogiannakis, Nikos Tsikoduis and Antonis Papaioannou that contributed for a pleasant and productive envrinment all these years in the lab and also Giannis, Eygenios, Fotis, Lazaros, George and Mavra. I would like to thank the Foundation of Research and Technology, Hellas for its support. Finally, I would like to thank my family who has always been there for me all these years all these years.

The research outlined in this thesis has been funded by the Foundation of Research and Technology Hellas and the TRACER Project (09SYN-72-942). I was able to complete this thesis, while I was supervised by both Professor Evangelos Markatos and Dr. Sotiris Ioannidis. The research has been conducted in the Distributed Computing Systems laboratory, which is hosted in the Institute of Computer Science of Foundation of Research and Technology, Hellas.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Controlling the flow of information is paramount to the security of applications. Web applications, in particular, pose a challenge to traditional information flow techniques, because they span a multitude of layers, platforms and languages. To control information flow in a web application, certain parts must be designed accordingly from the ground up, during the development cycle, to reflect the desired policy sets. Even then, web applications are composed of many parts, possibly written in different languages, making it difficult for the programmer to implement a security policy, test and debug it. For the same reason, changing an existing web application to control information flow or adhere to, for instance, a specific privacy policy, is very difficult.

Unfortunately, the majority of popular applications has not been designed with privacy as a prime consideration. Legacy applications are more susceptible to information leakages, which may lead to financial loss [13] or loss of users' privacy [9]. The cost of redesigning an application to harden its security may be prohibitively high, or the functionality of the system may be so important to its users, that they may be resistant to change.

Even when a security policy is designed into an application, it is the responsibility of the developers to implement it correctly. In essence, it is up to the programmer to find all the points in the code where e.g., sensitive data may leak and insert the appropriate checks. In large, complex applications that undergo continuous development, it is very easy to miss such a check, forget to patch all points, etc., often introducing information leaks, vulnerabilities and exploits.

For example, MediaWiki is a wiki application written in PHP, developed and used in Wikipedia and other online encyclopedias, dictionaries, etc. As such, it is designed to facilitate collaboration and information sharing, not avoid leaks and control access levels. Indeed, MediaWiki's manual explicitly states that:

> *"MediaWiki is not designed to be a CMS, or to protect sensitive data. To the contrary, it was designed to be as open as possible. Thus it does not inherently support full featured, air-tight protection of private content."* [16]

Changing such a complex application to implement various security policies is very tedious and error-prone, as the system was not designed to track and restrict information flow.

MediaWiki in particular, and web applications in general, usually follow a three-tier architecture consisting of client-side code, server-side code and a database. This multi-tier architecture [12] imposes an extra problem to correctly implementing and enforcing security and privacy policies, as the programmer has to reason about persistent state in the database, untrusted user input, arbitrary client-side code behavior, etc. Existing solutions for system-wide information flow [8] are often too general; they cannot take into account (*i*) the specific application semantics and policy requirements — causing false positives, and (*ii*) the distributed setting of a web application, where the database may very well be at a different machine —causing false negatives.

This paper presents LabelFlow, a system for dynamic information flow tracking on web applications in PHP. LabelFlow aims to improve security and privacy in legacy web applications using label-based information flow. LabelFlow is designed to handle the 3-tier architecture usually found in web applications; it transparently extends the database schema to associate information flow labels with every row; it extends the PHP bytecode interpreter to transparently track labels at runtime; and it combines the two so that the programmer need only implement the policy code with minimal or zero changes to the rest of the legacy application.

LabelFlow works in the PHP language runtime, implicitly tracking labels for every piece of data received from or sent to the user, and data written to or read from the database. LabelFlow does not specify explicit, fixed policies; instead it provides an API to the user to write the policy code, i.e., a mechanism to create labels and associate them to pieces of data. The programmer can then use this mechanism to implement and enforce a wide range of policies with minimal changes to the rest of the application code.

In comparison, the state of the art PHP data flow system is Resin [35]. In Resin, the developer writes application specific code for the assertions that must hold for each piece of data. Resin ensures the proper propagation and the timely execution of the assertions. Resin, however, requires the developer who implements the assertions to have detailed knowledge of the application implementation. In LabelFlow, the policy is expressed in an application agnostic representation, making the migration easier. Finally, LabelFlow is lightweight compared to Resin, imposing much less time and space overhead on the application. Overall, this paper makes the following contributions:

- We designed LabelFlow, an information flow framework for implementing security and privacy policies in legacy web applications. LabelFlow can be used in a wide range of web applications, with minimal programming effort.

- We implemented LabelFlow in the PHP runtime, targeting web applications that use MySQL for persistent storage. Our implementation is fast, imposing an overhead of 5% over the original PHP runtime.

- We formally defined LabelFlow's semantics for a simple language that abstracts over PHP, and proved that it protects against information leaks.

- We deployed LabelFlow in existing real-world applications. More precisely, with minimum code changes (less than 100 lines of code), we apply LabelFlow on MediaWiki, WordPress and OpenCart.

This thesis is organized as follows. In Chapter 2 we present a brief background in *dynamic information flow* and access control. In Chapters 4, 5 we present our solution. In Chapter 7 we present a brief survey of the related work. Finally, the thesis ends with some conclusions in Chapter 8. In Appendix A, we present in detail our language and its properties.

Chapter 2

Background

The most common security policy in web applications is *access control.* Such policies model every user of the system with an identifier and describe which data a user can access. Access control policies restrict the release of information, but not its propagation afterwards. Once the information is released, all control over it is lost. In contrast, *information flow* policies ensure that the propagation of data follows the specified policy. For instance, a policy may dictate (i) the users who could access the information and (ii) places in the code where the data can be used.

Information flow policies partition program variables into different security levels and restrict the flow of information among variables in different levels. *Label-based information flow*, in particular, uses a set of labels to represent security levels and to track the flow of information. Consider, for instance, the simplest two security levels *secret (H)* and *public (L)*. Program variables are assigned one of those *labels* —we write $X : H$ to denote that variable X has security level `secret`. To enforce the policy we must prevent for any variables $X : H$ and $Y : L$, any execution $Y := X$ that would consist an information leak, because the *secret* label is more restrictive than the *public* label. Information can propagate from L to H but not the other way around.

Note that labels can have different semantics according to context. Labels can be used to label secret or public data in one context and trusted or untrusted data in another context. A label-based information flow system like LabelFlow simply tracks the propagation of data and their labels as the program executes. Individual label semantics are defined by the programmer according to their needs and application policy. In general, one can implement many kinds of security and privacy policies using label-based information flow: access control lists, tainting analysis, public/private data, etc.

In the simple model with two labels, *secret* is more restrictive than *public* —we write $L \leq H$. Real-world applications may have multiple security levels in many contexts, so, their labels do not need to be in the same hierarchy. To support more expressive label dependencies, we use a label lattice [18]. The label lattice is usually a semi-lattice with the following properties. (i) A label l_1 is more restrictive than a label l_2 if there is a path from l_1 to l_2 in the label lattice. (ii) The bottom of the label lattice always represents the label with lowest restrictions. The lattice create a transitive, partial order relation between labels, better suited to represent policies in complex applications.

2.1 Non interference

Side channels, like time attacks [5, 38, 4], the program's execution flow, power analysis, etc., can also cause information leaks. To protect against such leaks, a secure information system must enforce the property of *non-interference*. Non-interference dictates that an attacker would not be able to distinguish two runs of the program if they differ only in their secret values. Unfortunately, full non-interference is too strict to be enforced in practice. Moreover, it is a property of all execution paths, i.e., it can only be enforced using static techniques. Dynamic systems cannot normally decide non-interference, as they only observe one possible execution path. In LabelFlow, however, we restrict secret values to the persistent database, which allows us to enforce a (somewhat relaxed) non-interference property dynamically.

According to the most widely used definition of *non interference*. A program C satisfies the property if, for any two different executions of the program with memories M, N that agree on *public* variables, the final memories after running C also agree on the *public* variables.

The usual threat model assumes an attacker that can observe only the *public* input and output of the program. An attacker cannot distinguish between runs of the program with different *secret* variables thus cannot gain any knowledge about those, *secret*, variables.

The notion of *non interference* provides some very strong security guaranties but it is considered impractical for real world applications. As an example, consider a user trying to login in a web application. The user's password are stored in the database with *secret* label. By comparing the user's input with the stored password we mark the result of the comparison as *secret*, since it depends on *secret* data. As a result we cannot return the result of the comparison to the user because it is marked as *secret* and the user, before signing in, can only see *public*. As another example consider a simple aggregate over *secret* values. Assume that we have stored in a database the salaries of all the employees in our company, certainly each individual salary is marked as *secret*, but we would like to publicly display the average salary. Using *non interference* the average salary is considered *secret* value, because it is a computation of *secret* data. In this thesis we have taken into consideration the above limitations and we provide a mechanism for controlled *declassification*.

$$
\begin{array}{llcl}
\text{(Constants)} & n & \in & \mathbb{N} \\
\text{(Base Labels)} & k & \in & \mathcal{L} \\
\text{(Labels)} & l, pc & ::= & k \mid x \mid l \sqcup l \mid \bot \mid \top \\
\text{(Constraints)} & C & ::= & \emptyset \mid C, l \sqsubseteq l \\
\text{(Values)} & v & ::= & l \mid n^l \mid () \mid \lambda x \,.\, e \\
\text{(Expressions)} & e & ::= & v \mid e\ e \mid \textsf{create table} \mid \textsf{insert}\ e\ \textsf{into}\ n \\
 & & \mid & \textsf{update}\ e\ \textsf{to}\ e\ \textsf{in}\ n \mid \textsf{newlabel} \\
 & & \mid & \textsf{taint}\ e\ \textsf{with}\ e \mid \textsf{elevate}\ e_p \\
\text{(Databases)} & DB & ::= & \emptyset \mid DB, T \\
\text{(DB Tables)} & T & \subseteq & \emptyset \mid T, (n, l)
\end{array}
$$

Figure 3.1: A simple calculus with dynamic labels and persistent state.

Chapter 3

Formal Semantics and Soundness

We formalize our changes on PHP using a simple calculus extended with database persistent state, we define a small-step operational semantics for our language, and state the theorem of correctness for label flow. The full details of the formal proof can be found in the Appendix A.

3.1 Labels

Figure 3.1 presents a simple functional language with support for dynamic labels and database queries. Base labels k are label "atoms", label representations created using our dynamic label API. Any combination $l_1 \sqcup l_2$ of labels is also a label. The label lattice C is a set of $l_1 \sqsubseteq l_2$ constraints among labels. Values include unit, functions, all labels l and integer constants n^l, where we annotate the integer value n with its run-time label l, to reflect the run-time behavior of our PHP VM. All constants in the program code

$$\text{E-New}\ \frac{l - \text{fresh}}{\langle DB, pc, \textsf{newlabel}\rangle \to \langle DB, pc, l\rangle}$$

$$\text{E-Create}\ \frac{}{\langle DB, pc, \textsf{create table}\rangle \to \langle (DB, \emptyset), pc, ()\rangle}$$

$$\text{E-Insert}\ \frac{T'_k = T_k, (v, pc)}{\big\langle T_1, \ldots, T_k, \ldots T_n, pc, \textsf{insert}\ v^l\ \textsf{into}\ k\big\rangle \to \langle T_1, \ldots, T'_k, \ldots T_n, pc, ()\rangle}$$

$$\text{E-Select}\ \frac{T_n \in DB \qquad (v, l_2) \in \{(v, l) \mid (v, l) \in T_n \wedge l \sqsubseteq pc\}}{\big\langle DB, pc, \textsf{select}\ v^{l_1}\ \textsf{from}\ n\big\rangle \to \big\langle DB, pc, v^{l_2}\big\rangle}$$

Figure 3.2: Selected semantic rules

are trivially annotated with the label $\bot$. Another special label is $\top$ which is the most restrictive label of the language.

3.2 Syntax

Program expressions e include function application, database primitives and dynamic label allocation. Intuitively, expression create table creates a table in the database, expression insert e into n inserts the result of expression e into the n-th table of the database, expression update e_1 to e_2 in n updates table n, replacing any row that is equal to the result of e_1 with the result of e_2, expression newlabel creates and returns a new label at run time, expression taint e_1 with e_2 computes e_1 to an integer and e_2 to a label, and taints the integer with the new label, and expression elevate e_p computes expression e_p (which should not have side effects in the database) to a label, and sets the current state, pc, to that label. During the execution of e_p the pc is set to $\top$, thus it is has unlimited privileges.

3.3 Operational Semantics

Figure 3.2 presents a subset of the small-step operational semantics for the language. Judgments have the form $\langle DB, pc, e\rangle \to \langle DB', pc', e'\rangle$, where DB is the database state, pc is a label representing the "current elevation" level, and e is the executing program. After the program takes a step to e', the database may have changed to DB' and elevation level pc'. Rule [E-New] executes the dynamic creation of a label, where expression newlabel always takes a step to a fresh label l, not previously occurring in the database. Rule [E-Create] creates an additional table in the database, initially empty of rows. We abstract over table names and database row fields, instead using the table creation order n to identify database tables in all queries, where every table has only one column containing values, and a column holding the label of every row. Rule [E-Insert] inserts a value v into the database, using label pc. Finally, [E-Select] shows the execution of a select query which verifies that the value selected is visible in table n using the current pc elevation.

3.4 Soundness

We use the semantics to prove that any code not using $\mathsf{elevate}\ e$ instructions satisfies noninterference, i.e., cannot leak any data labeled by a label above its pc. To do that, we define the following:

Definition 1 (Table Similarity). *Let tables $T_1, T_2 \subseteq \mathbb{N} \times \mathcal{L}$. We say that T_1 and T_2 are similar up to l and write $(T_1 \sim_l T_2)$, if $\forall l' \sqsubseteq l, v\ (v, l') \in T_1 \Leftrightarrow (v, l') \in T_2$.*

Definition 2 (Database Similarity). *Let databases $DB_1 = \{T_1, \ldots, T_n\}$ and $DB_2 = \{T'_1, \ldots, T'_n\}$. We say that DB_1 and DB_2 are similar up to l and write $DB_1 =_l DB_2$ if: $\forall 1 \leq i \leq n, T_i \in DB_1, T'_i \in DB_2 \Rightarrow T_i \sim_l T'_i$*

The above definitions define the similarity between two databases. This definition is similar to the standard definition of *non interference* requesting that two memories are similar. In practice two databases are similar for a user holding the l label, if they are identical for the values having labels less restrictive than l. In essence, its user is only viewing a "projection" of the database containing only the values he can access.

Theorem 1. *Assume e is an expression without any $\mathsf{elevate}\ e$ terms, l and pc are labels, and DB_1, DB_2 are databases with $DB_1 \sim_l DB_2$. Then executing e under the two different databases with input labeled l will yield the same results: $\langle DB_1, pc, e\rangle \to^* \langle DB'_1, pc, v\rangle$ if and only if $\langle DB_2, pc, e\rangle \to^* \langle DB'_2, pc, v\rangle$ Moreover, it will be $DB'_1 \sim_l DB'_2$.*

Chapter 4

Design

LabelFlow aims to integrate easily with existing web applications, with minimal changes. LabelFlow protects sensitive information inside the application from reaching unauthorized users by malicious actions or programming errors. We target web application with a 3-tier architecture, where the *presentation*, the *application* and the *storage* are three distinct components running on different platforms, as seen in Figure 4.1.

The presentation tier is inherently unsafe since it is executed in the user's browser. Sensitive data should not reach the presentation layer of an unauthorized user, as this amounts to an information leak. It is very easy to intercept the information on the wire or modify the client code to steal the information. Information is only safe so long as it stays in the application or the storage tier. One of the challenges in this work was to ensure that labels propagate correctly when data migrate between the *application* and *storage* tier. In this chapter we describe the components of LabelFlow.

4.1 Application Layer

The application layer implements the core logic of the application. The application receives input from the presentation and compute, in collaboration with the storage layer, the appropriate response. The result of the computation can modify the presentation layer, if a response is rendered back to the user, and/or the state of the storage layer, if it triggers modification of the database.

Dynamic information flow is a versatile tool that can implement different security and privacy features. The main focus of this thesis is on access control that protects against unauthorized access to data but other features to protect against code injections are also possible.

Initially, the programmer must label sensitive data that need to be monitored using our API. Deciding which data needs labeling depends on privacy policy the developer wishes to enforce. For instance, if the developer wishes

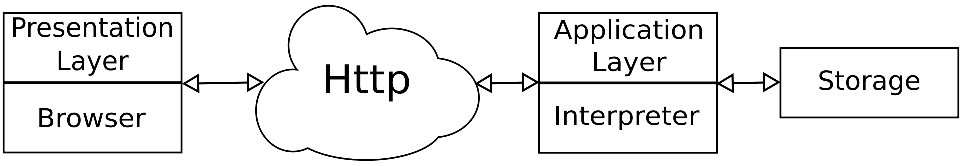


Figure 4.1: The architecture of typical Web application

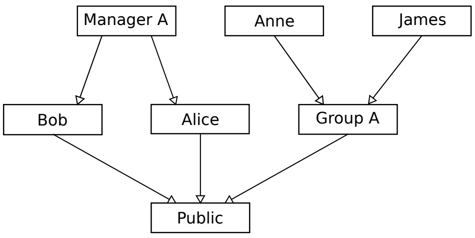


Figure 4.2: An access policy described as a label graph.

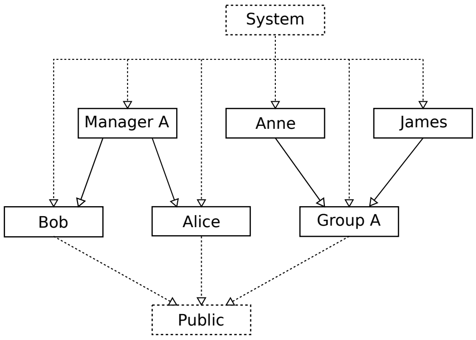


Figure 4.3: The full label graph used by LabelFlow. The graph is a full-lattice, the graph always have a `public` and a `system`. All labels are more restrictive than `public` and less restrictive than `system`.

to enforce an access control policy, they should create a label for every user and associate new data with the labels representing only the users that can access it. One such policy we present in Figure 4.2. Alternatively, implementing a tainting analysis needs only two labels for trusted and untrusted data.

Apart from initial labeling, the application should follow its normal execution path. During execution, data values that depend directly on labeled data are also transparently labeled. If two operands have different labels the result is labeled with a combination of those labels (usually the union of the labels). Chapter 5 discusses propagation in detail.

We assume that the application can distinguish between the users. It is common for Web applications to require that the users login using a password. The application then associates a 'cookie' with the user to identify him in subsequent requests. The mechanism that identify the user whose request we are currently serving should be extended to retrieve the appropriate label of the user. That label is stored in a variable called *program counter (pc)* and it is used to evaluate what data from the database are accessible during this execution. To reduce the risk of an arbitrary change of the *pc* either from a programing error or from a code injection, *pc* can only change inside

```
INSERT INTO table_name (name1, name2, name3) VALUES ('bob', 'alice', 'anne')
```

```
INSERT INTO table_name (name1, name2, name3, ac_label) VALUES ('bob', 'alice', 'anne',
Bob⊔Alice⊔Anne)
```

Figure 4.4: Example of inserting into the database data with different labels. The strings `'bob'`, `'alice'` and `'anne'` have labels `Bob`, `Alice` and `Anne` respectively. The exact SQL writing mechanism is described at Chapter 5

an `elevate` statement. To defend against code injection attacks, where the attacker calls `elevate` from an crafted string using `eval` we have disabled the `elevate` statement inside `eval`.

4.2 Storage Layer

Almost all modern web applications use some kind of persistent storage. This need is further exasperated by nature of HTTP. HTTP is designed to be stateless [29], so web servers are designed to process each request in a new and isolated environment.

Applications, on the other hand, need to maintain state. Information about users, like their name, passwords, or credit card numbers, information about the state of the application, like number of users, statistics, or content, need to be reliable stored for future access. Most web applications use relational databases for their back-end storage, because there are several mature implementations and extensive support from the programming languages.

A database being an important component of any web application, data should not lose their labels when stored in the database. Otherwise, labeling is not persistent across requests. Storing this additional information in a database is difficult to do manually, because it requires modifying the schema. LabelFlow automatically extends the database schema with a label per row, for each table. This granularity is similar to row-level security offered by several databases (Oracle, IBM, Microsoft), and means to label the data forming the row, but also their relation.

Our approach requires specific changes to the database schema of the application. This, however, is not trivial to do manually, as the schema may be dynamically generated according to installation configuration options. Installing web applications is commonly done via their web interface, so it often uses the same database API to send `CREATE TABLE` queries to the database, as it does for common selection and update. Thus, we have designed LabelFlow to intercept the queries from the application to the database at run time, and automatically rewrite them to change the schema as necessary, transparently adding a label per row in each table. We opted for this method instead of changing the schema after installation, as done by systems in related work, because (i) installation and creating a schema is a part of the web application, and thus may leak information, and (ii) it makes porting a web application to LabelFlow easier.

We decided to restrict granularity to a label per row of each table, instead of the finer granularity of a label per field [7]. LabelFlow extends each table in the database with an extra column where the label is stored. Certainly, a finer-grained granularity allows for more control over which information is tainted with a certain label. However, coarse-grained labeling per row reduces space requirements and minimizes changes to the original schema. Moreover, the relation among data items may be important. For example, consider the case where even though two pieces of information are public, their relation may be secret. To capture such cases, we use one label per row.

Moreover, row-based labeling allows for easier and faster query rewriting. To guard against information leaks when a row consists of fields with different labels, we use the following conservative policy: The label of the whole row is the "meet" of the labels of all fields stored in the row, as in Figure 4.4. This conservative policy can restrict the label of some fields even further, when, for example, many public data items are stored in the same tuple with a secret data item protects against data leaks.

4.3 Label Graph

Consider the secure MediaWiki application example described in Chapter 1. MediaWiki users generate data, which they may wish to keep private from or share with other users. The generating user is the owner of the new data and he should be able to choose the privacy policy regarding his data. LabelFlow provides a powerful and application-agnostic mechanism to express privacy policies.

Overall, in addition to labeling new data, the application programmer can use the LabelFlow API to add "sub-label" edges among labels, essentially structuring all labels into a semi-lattice. We use the semi-lattice model proposed by Mayer et al. [18], where there is an reflexive, transitive, *acts-for* partial order relation between the labels. The semi-lattice includes an implicit, common "bottom" element for all labels regardless of their context, so that LabelFlow can use it as a default label for otherwise unlabeled data. Normally, this "bottom" label in the semi-lattice corresponds to public information, every user in the system, etc., according to the policy implemented.

The owner can choose to create a fresh label inaccessible from everyone to keep their data private, use the "bottom" label to freely share data, or assign a label accessible only from a small group of other users. With this model, the owner of the data can grant access to any combination of users. Note that implementing the graph requires knowledge of the desired policy and of our framework; it does not require detailed knowledge of the application. We believe this is important for legacy applications where continuous iterative development may have rendered the code base unreadable.

Figure 4.2 shows an example label hierarchy for a hypothetical instance of the MediaWiki application. The vertices are labels and directed edges correspond to the partial order relation. The *Public* vertex is the "bottom" element of the semi-lattice. In general, an edge between labels A and B captures the relation A *acts for* B, meaning that label A is more restrictive than label B. Labels *Anne, James, Bob* and *Alice* are unique to their respective users, whereas, labels *Manager, Group A* and *Public* were created to facilitate sharing between the users.

In Figure 4.3 is the extended label graph generated by LabelFlow based on the graph provided by the developer. The graph is a full-lattice with two distinguished nodes *public* and *system*. The *public* is the bottom of the lattice and represents data that are publicly available. For convenience the *public* defined by the application and the one generated by LabelFlow merge. The *system* is the top of the lattice, by construction is always above any user-defined label. Consequently, someone having the *system* label can access all data of the application.

4.4 Sign In

As discussed in Chapter 1, in Web application it is desired to leak an amount of information. One such situation is during the login of a user. Prior to a successful login the user can only access only public content of the application, LabelFlow ensures that by setting the *pc* to *public* by default. Authenticating the user is out of the scope of this thesis, we just assume that the application has the necessary authentication code. Web Applications are common to user a password based authentication, the user provides a passwords which is checked against the stored password for matching. This is a form of declassification because while executing code in *public* context, the passwords are inaccessible. To allow this form of declassification LabelFlow supports the `elevate` expression. The developer should call the authentication code inside the `elevate` statement.

Chapter 5

Implementation

This section describes the implementation of LabelFlow. To implement dynamic, label-based information flow, LabelFlow is comprised of three components: (i) support for label-based information flow in the PHP runtime engine and standard library, (ii) support for transparent rewriting of database queries to include labels, and (iii) a library of PHP code that exposes the LabelFlow API to the web application programmer, as well as implementations of common policies.

5.1 PHP Runtime

To track information flow in the PHP part of the application, we modified the PHP runtime engine to propagate labels along with data. This approach is transparent to the PHP programmer and does not require any dynamic or static rewriting of PHP code. The LabelFlow modified PHP runtime engine is based on a prototype engine by W. Venema [30], designed for defending against well known web attacks such as Cross-Site Scripting and SQL injection using runtime taint analysis. That runtime engine can prevent such attacks by marking data coming from the network as untrusted, potential leading SQL or HTML injections, or PHP control hijacking. The engine tracks untrusted data, which cannot be used by certain function calls without prior sanitization. We ported this runtime engine to a current version of PHP, as it was unmaintained, and extended it with support for generic label propagation, additional primitive operators, and foreign function calls.

The PHP interpreter, named the *Zend* engine, is written in C. The runtime engine parses PHP code and generates a series of opcodes which are then executed. The opcodes are in an intermediate bytecode representation between the PHP code but higher-level than assembly language. The PHP runtime engine represents userspace variables internally as values of type *zval* struct. We extended this structure with an additional field, the labeling field, where the labels of each value are stored.

5.1.1 Label Representation

We use a bit-vector representation for labels, where the taint field is 32 bits long; we use one-hot encoding to represent the labels, thus our system can support up to 32 labels. The number of labels is limited but easy to extend at minimal cost. Additionally, one-hot encoding of the label permits very fast manipulation of the taint bit using bitwise operations.

We propagate labels on value copy by copying the taint field from the origin value to the destination value. Similarly, we have added support for all internal PHP arithmetic, string, bitwise, copying, assignment and update operators, so that the resulting value is labeled appropriately. When the operands have different labels, we label the resulting value using both, meaning that in the bit-vector representation two bits will be enabled. Note that we do not conflate labels even when they have a "meet" label in the label graph.

5.1.2 Foreign Function Interface

Unfortunately, the original implementation of taint propagation in the PHP runtime engine that we used, does not work with calling functions implemented in a third language. This is a problem, as the default PHP runtime engine is bundled with a rich set of standard functions called the *standard API*. Their functionality ranges from string processing functions to database interfaces. These functions are implemented in C for speed and thus do not use the PHP operators to propagate labels from operands to results. A possible solution would have been to manually modify each of these functions to copy the labels of their parameters to their return value. Although possible [35], this solution is laborious and thus error prone. It also requires in-depth understanding of the semantics of each function so that the right labels are returned. Moreover, if more functions are later added to this standard library, it is up to the developer to implement label flow propagation in the new extended function set. For the above reasons we implemented the following alternative solution. For all functions that belong to the standard library, the return value is conservatively labeled with the union of the labels of the arguments used when the function was called. Moreover, to protect against functions that return values by changing the state of their arguments, we also label each argument with the union of all labels of the arguments. This is potentially a very conservative approach, but it ensures that no information leak will happen from the execution of the function. Since we cannot track the information flow inside the function, we assume each argument could have tainted each other argument or the return value.

5.2 Database Modifications

Web applications almost always use persistent storage, where they reliably store information essential for their normal operation. This storage is normally a relational database. Currently, LabelFlow works with the MySQL database. To store extra information in the database we need to extend the schema with extra fields where the labels can be stored. We believe that a reasonable trade-off between accuracy and space on one hand, as well as easy-to-implement and easy-to-manipulate on the other, is to store a label per row. That means that all the fields in the same row are stored under the same label, even if during execution their labels were different. To ensure that there is no information leakage, we conservatively set the common label to be the union of all labels of all fields of the tuple.

All the necessary modifications in the database schema and in the queries inserting and retrieving data from the database take place by automatically rewriting the corresponding queries. To extend the schema the `CREATE TABLE` queries are also rewritten to have one additional column. The `INSERT` queries populate that column and the `SELECT` queries retrieve it. We use a custom SQL parser written in C to parse and modify all database queries at run time, including the creation of a new schema during the installation of the application.

Figure 5.1 shows a representative example of SQL rewrites. The first query shown in Figure 5.1(a) (lines 1–6) originally creates a table with three fields. LabelFlow intercepts the query and rewrites it as shown in Figure 5.1(b). The `CREATE TABLE` query is rewritten to include an extra field to store the label for each row, shown in Figure 5.1(b) (line 6). The second query shown in (a), lines 9–11, inserts a tuple in the table. LabelFlow rewrites this to also insert a value in the label field, shown in (b), lines 9–11. The label value corresponds to the union of all fields' labels. Finally, the third query performs a selection on the table. We rewrite this to also constrain the row label to the label of the user performing the query. Effectively, this creates a "view" (projection) of the table depending on the label used to generate the selection query. Note that we have used the equality test, and a predefined user label in the example for the sake of simplicity. Normally, the rewritten query tests for any label *up to* the label used to perform the query, which can be an arbitrary label depending on the policy implemented by he application.

5.3 LabelFlow library

LabelFlow is implemented as a set of PHP functions and classes that are easy to incorporate into the application. Specifically, LabelFlow provides the following functionality: (i) A high level API where each application can

```
CREATE TABLE
( fname   VARCHAR(100),
  lname   VARCHAR(100),
  address VARCHAR(255))


INSERT INTO table_name
  (fname, lname, address)
  VALUES (1, 2, 3)

SELECT fname, lname, address
  FROM table_name
  WHERE condition
```

(a) Original SQL code

```
CREATE TABLE
( fname    VARCHAR(100),
  lname    VARCHAR(100),
  address  VARCHAR(255),
  label_ac SET(...) default 1)

INSERT INTO table_name
  (fname, lname, address, label_ac)
  VALUES (1, 2, 3, label)

SELECT fname, lname, address, label_ac
  FROM table_name
  WHERE ((label_ac | user_label)=user_label)
          AND (condition)
```

(b) SQL code after rewriting

Figure 5.1: Example SQL queries, rewritten by LabelFlow.

register meaningful names as labels, (ii) an API for constructing the label graph discussed in Section 4.3, and (iii) a custom database API.

Internally, the PHP engine encodes labels as integers stored in internal data structures. This encoding may be efficient but is very cumbersome to use in real applications. Also, it is better if the internal representation of the labels is hidden from the application to minimize hijacking attempts. At any given moment the LabelFlow stores the *program counter label, pc*. The *pc* is the context under which the system should evaluate its policy. Normally, when a user logs in the application the *pc* is set to the user's label. The *pc* defines a privacy context that is taken into consideration regarding which data should be accessible or not.

The database API has the same interface as the default PHP API and thus migration is an easy task. This API is responsible for rewriting the SQL queries and for supporting the persistent labeling of the data. On `CREATE TABLE` queries it injects the extra table in the query. On `INSERT` queries it retrieves the label of the query string and calculates the label value to be written in the DB. On `SELECT` queries LabelFlow performs two operations. First, it ensures that only data accessible by the user who initiated the request will be returned from the database. The results that are accessible must have less restrictive labels than the current *pc*. This is a security mechanism protecting unauthorized access to data. Second, the returned data are re-labeled to ensure proper label flow control.

In most applications, the PHP engine usually terminates after serving one request and restarts to serve the next one. It is hard to hold information in the engine itself. For that reason, LabelFlow needs access to the database for storing in two tables the mapping between the application-level representation of the labels and the low-level integer representation. It is important to store the mapping for avoiding registering the same integer under a different name in a subsequent call. The second table holds the label graph.

The typical steps to integrate LabelFlow in an existing legacy application are the following.

1. Incorporate LabelFlow with the application we want to apply flow control. This action involves including the source file of our framework in the main file of the application and instantiating the LabelFlow object. The LabelFlow object accepts as parameters the credentials to a database for storing its internal tables.

2. Replace the database API calls with our wrappers. Since our wrappers have the same signatures as the standard functions, this action is done automatically.

3. Define the principals and the label graph.

4. Generate and store a meaningful label for each principal of the application. Principals can represent users, groups, or roles, according to the application's needs. The application is responsible for storing the label with each principal and retrieve it when it is executing actions on their behalf.

5. Call the label function to label incoming data in all application entry points. By default, the data will be labeled with the *pc* attribute. We assume that the application has correctly authenticated the user and has assigned the *pc* the corresponding label.

5.4 MediaWiki

MediaWiki's modular and object oriented architecture facilitates migration to LabelFlow. Including our code to the project and initializing LabelFlow is simple, and requires adding just two lines of code to `Setup.php`. All data received by the user pass from a central point where they populate PHP structures for easier processing. At that point we label the data with the *pc*. Most applications have a limited number of well-defined entry points that makes it easier to label data as they arrive. We used the built-in mechanism to store the labels of each user. We manually extended the table *user* where the application holds information about the registered users, like their name, their password etc., to also contain the labels that have been assigned to them. At the PHP level the users are modeled by the *User* class. We extended that class to store and retrieve the label of the user as it happens by default with all of the user's attributes.

When a user edits an article, that article is labeled with the user's label, so that when the SQL query is constructed to save the article in the database, the label is further propagated in the query string. Later, when the query is transferred to the database the label is also stored. One issue we encountered in MediaWiki is that the different revisions of an article are stored in a linked list. The head of the list is the latest revision and each revision holds a pointer to the previous one.

This can lead to a problematic behavior: if a user does not have authority to access the latest revision then the link to the previous revision, which may be originally accessible, has been also lost. To solve this problem we where forced to change the schema of the MediaWiki application. Currently, MediaWiki holds an entry in the *page* table containing the title of the article, some statistics and a pointer to the latest revision of the article. Information about the revisions of the article are stored in the *revision* table. The *revision* table stores information like the time of the revision, the user who made it, a pointer to the previous revision of the article and a pointer to actual text of the article at the current revision. The text of the article is stored in a table name *text*. Because the insertion in the *page*, *revision* and *text* tables inserts data labeled by the user who made the changes, other users will not be able to retrieve that information in the database. That may ensure privacy but greatly reduces the functionality, since it is not the desired behavior. To achieve the expected behavior we made some modifications that white-list the *page* table, so that all users can have access to the list of articles. This may leak some information but we believe it is acceptable. Additionally, all revisions in the *revision* table of an article hold a pointer back to the page entry of the article. This allows to locate previous revisions of the article even if we do not have access to the latest revision. After those changes, which where less than 50 lines of code, each user has access to the latest revision, according to their label.

Chapter 6

Evaluation

To test the engine overhead we used *bench.php*, the standard benchmark bundled with the engine, namely a loop of CPU intensive operations, and thus closer to the worst case than typical workloads. While the unmodified engine takes an average (over 10 runs) of 21.4 seconds, the LabelFlow engine takes an average of 22.6 seconds, i.e., LabelFlow causes 5.6% overhead.

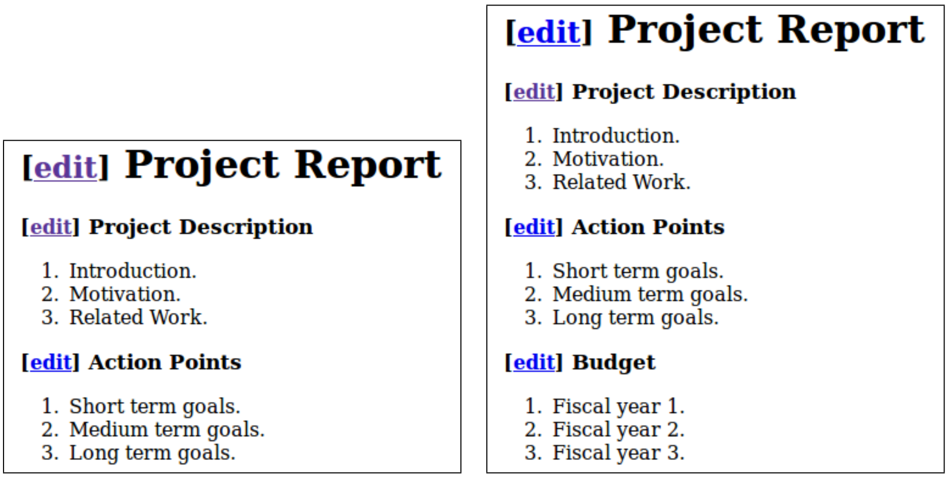
To test LabelFlow's applicability and ease of use, we used three widely used applications: MediaWiki, the wiki used by Wikipedia; WordPress, a blog hosting application; and OpenCart, an e-commerce and store management application. We run all experiments on a Pentium 4, 3.4GHz workstation with 3 GB of memory running Linux 3.0.0-17.

6.0.1 MediaWiki

In MediaWiki, users modify the articles and create new revisions. Using LabelFlow we implemented an access control policy where each user that creates a revision labels it with his credential. Other users who wish to read the article have access only to the revisions accessible from their credential.

For instance, Figures 6.1(a) and 6.1(b) show an article as viewed by two different users. The article is a progress report about a project. The first user 6.1(a) is a contributor to the project with low level clearance, and thus, can edit the details about the scope and the goals of the project and their changes will affect all other users accessing the articles. The second user is a high-level manager in charge of the project. The manager has higher level clearance, which allows them to see and edit the whole article, including the budget section. The budget section includes sensitive information about the economics of the project that should be kept secret. Any changes done by the manager in this article are going to be visible only by the users that have equal or higher level access than the manager. Those users will have labels that are more restrictive than the ones assigned to the manager, corresponding to "higher-up" in the label lattice.

The MediaWiki page provides a set of common security limitations [16].



(a) View of project member (b) View of project manager

Figure 6.1: The same page of our wiki as seen by two different users with different authorizations.

For some, MediaWiki offers suggestions on how to overcome them. We focused on the ones that offer no such suggestions (see Table 6.1). To the best of our knowledge LabelFlow is able to offer protection against all of these vulnerabilities.

The necessary modifications to enforce the policy on MediaWiki were fairly straightforward, totaling around 100 lines of additional code in a code base of over 100,000 lines. Moreover, they were often made apparent by helpful error messages while migrating to LabelFlow, when MediaWiki encountered an error. We measured the overhead that our changes impose to MediaWiki. To study the cost that our modifications have on the end-user experience, we measure the time needed to login and load an article, two representative operations. We performed 200 requests of each and measured response time.

Figure 6.2 (a) shows the time needed to log into the application. The login operation requires a database query to retrieve the information of the user and check that the password is correct. When no user is logged in, LabelFlow labels all data as *public* and performs all operations using the *public* label. The "public" label is a hard-coded value designed to represent the bottom of the label graph. All users can read data having the *public* label, but any user using the *public* label to request data can only see public data. Figure 6.2 (b), shows the total time needed to retrieve an article from the database. MediaWiki must retrieve the user's information based on their cookie, find the appropriate revision for the particular article for the user and finally retrieve the text. In both experiments, LabelFlow imposes only a small overhead, because of its efficient label representation and fast query

Type	**Vulnerability**	**Fixed with LabelFlow**
API	Can the *revids* parameter for action=query be used to fetch revisions that should be hidden?	Yes
Author backdoor	Some extensions always allow the original author of a page to access it, ignoring later access restrictions.	Yes
Redirects	Some extensions always allow the original author of a page to access it, ignoring later access restrictions.	Yes
Other extensions	Can a user use other extensions to view part of a page? Think of DynamicPageList or Semantic MediaWiki, which provide ways to query the database for certain pages or properties.	Yes

Table 6.1: Common Vulnerabilities

rewriting.

6.0.2 WordPress

WordPress is a popular open source blogging tool based on PHP and MySQL. In contrast to MediaWiki, WordPress offers an extensive set of roles ranging from *Administrator*, who has complete control over all aspects of the application, to *Subscribers*, who can only control their profile. Moreover, blog authors can limit the visibility of their profiles to selective users of the application. We used LabelFlow to enforce this policy on WordPress, and compare it with the native implementation. We noticed that the existing system has one limitation:

> *"WARNING: If your site has multiple editors or administrators, they will be able to see your protected and private posts in the Edit panel. They do not need the password to be able to see your protected posts. They can see the private posts in the Edit posts/Pages list, and are able to modify them, or even make them public. Consider these consequences before making such posts in such a multiple-user environment."* [33]

In WordPress, users do not create accounts for themselves, they instead rely on the administrator to create the accounts for them. Thus, initially the administrator must have access to user data, but should drop it as soon as possible. We encoded this behavior by having the administrator code create a new label for the new user, use it to taint all user data and then delete the label to make it inaccessible to the administrator. In total, to integrate LabelFlow into WordPress, we added 60 lines of code.

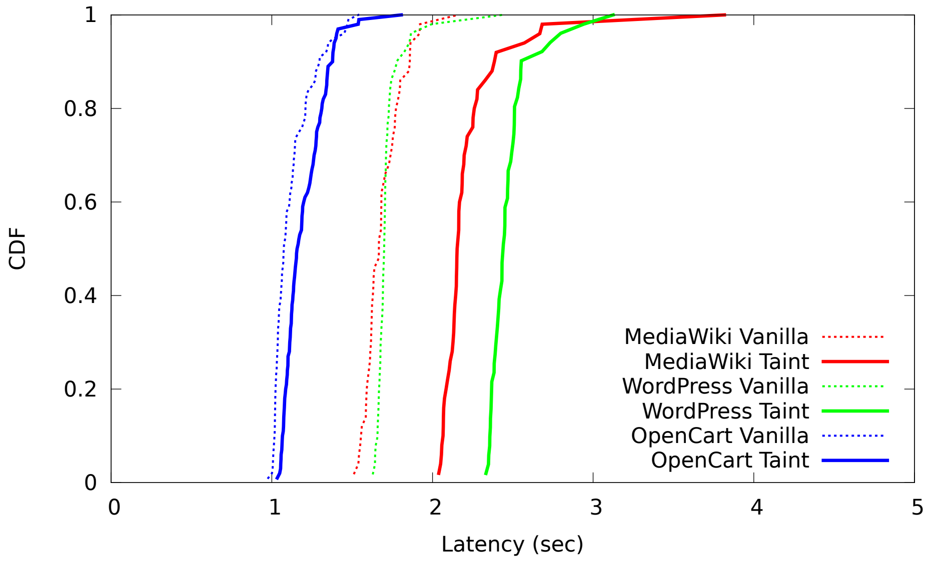
6.0.3 OpenCart

OpenCart is an e-commerce and online store-management software program. In OpenCart, system administrators add products available for purchase, and customers place orders and write reviews about the products they have purchased. OpenCart follows the MVC pattern, where the code is seperated into three categories: Model, View and Controller. Model is the database abstraction layer, View is responsible for the presentation of the information to the user and Controller implements the application logic. This architecture made it even easier to integrate LabelFlow into the application. It was easier than the previous two applications to identify the places in the code base where changes were needed. We were able to integrate LabelFlow easily in less than 60 lines of code, so that an administrator could limit the visibility of products to a audience of their choice.

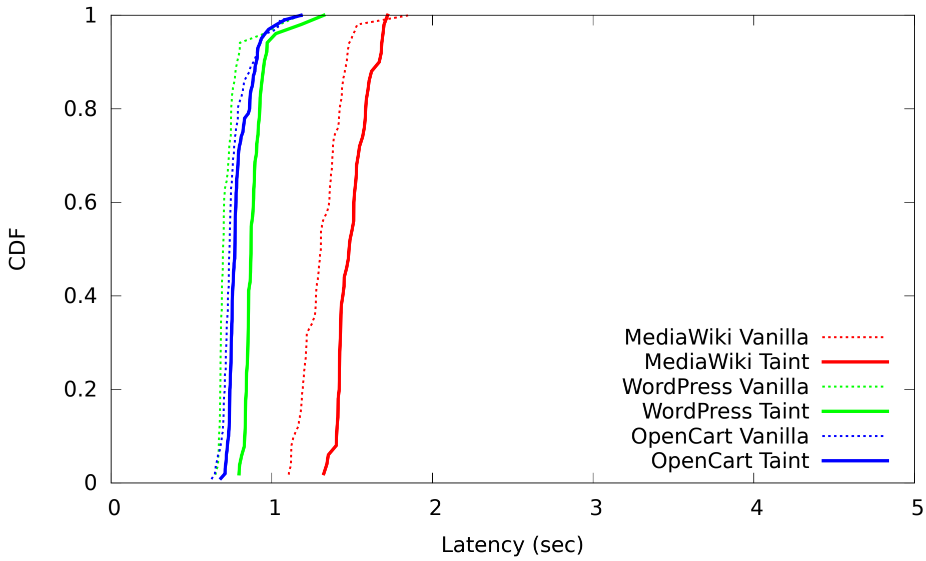
6.0.4 Comparison with Resin

Resin [35] is an information-flow system for PHP that uses assertion-based data flow. Assertions are pieces of code that implement the desired security or privacy policy for each piece of data. From a programmer's perspective, writing such assertions requires deep understanding of the application, its execution paths and its data structures, since the assertions are application-specific pieces of code. In comparison, implementing security and privacy policies in LabelFlow requires knowledge of the framework rather than the application, our policies are more *application-agnostic*.

Yip et al. report a performance overhead of 33% running Resin in the HotCRP conference management application. LabelFlow incurs a much lower overhead on running MediaWiki (see Figure 6.2). To further compare the performance of overhead of Resin and LabelFlow, we run a series of microbenchmark tests for both on the same system. Figure 6.3 presents the results. We have compared Resin, LabelFlow and their corresponding "original" versions of PHP. For Resin, the original version is PHP5.2.5; for LabelFlow, it is PHP5.2.17. Overall, we found that LabelFlow is significantly faster on SQL operations.

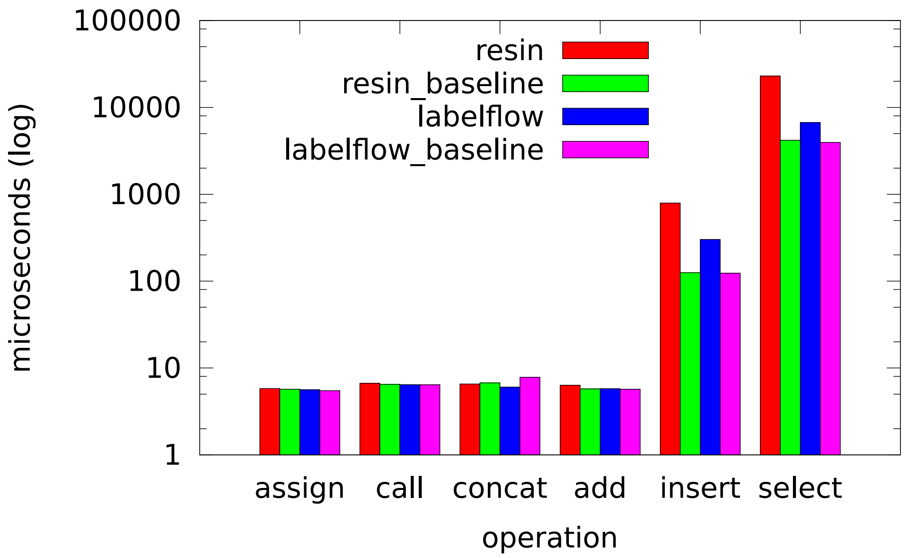


(a) User login.

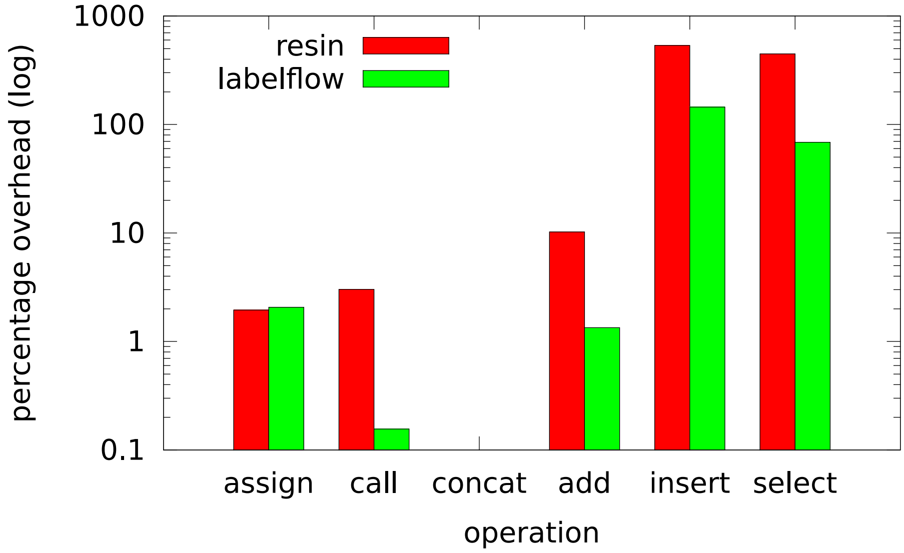


(b) Full page load.

Figure 6.2: Cumulative Distribution Function (CDF) of the time for two kinds of requests.



(a) Microbenchmark performance.



(b) Overhead imposed by each system over its baseline.

Figure 6.3: Comparison between LabelFlow and Resin.

Chapter 7

Related Work

In this chapter we review the related work. Since, we have borrowed ideas from from different fields of research, we have structure the section in many parts.

7.1 System Security

Tainting analysis [34] and flow tracking are both very active research fields. The academic literature is rich. The closest research effort to LabelFlow is Resin [35] and TBTaint [7].

Resin [35] is a language runtime that supports dynamically checking assertions in PHP and Python programs. Resin is designed to assist the developer to enforce the security policy they have in mind. In Resin the programmer write `policy objects` for each piece of information. Each `policy object` is a piece of code describing the policy the programmer wishes to associate with that information. A modified runtime ensures that the policy code will be executed before the information leaves the environment. Resin requires that the programmer has a thorough understanding of the application and also of the policy wishing to enforce. Moreover, Resin does not offers guaranties about information leaks.

TBTaint [7] provides cross-applications information flow tracking by extending the databases to store the tainting with the data transparently. TBTaint performs automatic query rewriting, like our system. TBTaint allows more fire-grained control over the tainting by storing tainting per value, instead of per row. TBTaint does a compatible runtime environment to work with the database. Applications must by aware about the existence of TBTaint to take advantage of it or a third party taint runtime can be used in co-operation with TBTaint.

TaintCheck [22] is a automatic detection that can detect overwrite attack. *TaintCheck* taints the input data of a program and the data derived from them and detects attempts to use the tainted data in suspicious ways. For instance to overwrite the return address or function pointers. *TaintCheck* can analyze binaries without any modification and has no false positives.

In the context of Web security Philip Vogt et. al. [31] proposed the use of dynamic and static taint analysis to protect against XSS injections. They propose a client system that keeps track of sensitive data e.g. cookies and blocks them from leaving user's computers.

WASC [21] is a complier for Web applications that automatically insert checks in the code to protect against SQL and script injections. WASCis build on top of GIFT a tracking service.

However, all of these frameworks target very precise problems, such as cross-site scripting [31] and SQL injection, or apply selectively to an isolated layer of the complete system. For example, tainting is used to investigate if the DOM of a web page has been infiltrated by foreign data [20]. LabelFlow follows a generic approach for enhancing web applications with information flow capabilities.

Xu et al. [34] propose automatic source rewriting to enhance program in C with dynamic information flow. They're approach of tainting memory positions instead of variables ensures correct propagation even when aliasing is used. Even though they're basic technique is sound, it suffers from unacceptable overhead, the optimizations they implement improve performance considerable but break the soundness. Moreover, they tainting is elementary only two values are allowed (taint-untaint).

Nguyen-Tuong et al. [23] was one of the first to propose taint propagation for Web application. Their approach is limited only in the PHP runtime, they do not take into consideration the database. In addition the proposed solution only protects against SQL injections and XSS attacks, it is not a generic framework.

Sekar [24] proposed another technique to emulate information flow in web applications. Sekar's approach aims at recognizing the use of untrusted input as output. Sekar approach modifies the Apache server to intercept input (Web requests) and output (Web responses and SQL queries), using *approximate string matching* the system tries to associate tokens of the input that appear in the output. Sekar chooses to emulate information flow instead in implementing it to reduce instrumentation of the different components of the Web application.

Sridharan et al. [25] proposed a technique to improve taint analysis in Web applications built using Web frameworks. The motivation behind this work is that the use of Web frameworks make use of reflection that can cause problems in static analysis and generate false positives. The authors propose a new specification language named *WAFL*. Developers will specify the framework related behaviour in *WAFL* to assist the taint engine distinguish between the application logic and the framework. This approach requires that a specification must be written for each framework and also existing taint engines must be augmented to understand the *WAFL*

Yumerefendi et al. [36] proposed another method for reducing information leaks. The proposed method detects privacy leaks using duplicate execution. Each time a process tries to read sensitive data, the operating system creates a second process, a *doppelganger*, the *doppelganger* process is similar, but not identical, to the original process. The kernel feeds the original process with the sensitive and the *doppelganger* with scrubbed data. The kernel monitors the execution paths of the two processes by comparing the system calls of the two processes. If at some point the system call diverges it is an indication that the execution path depends on the sensitive data. If the object that the system call modifies is insides the system jurisdiction the kernel mark the object as tainted and continues execution. If the object is outside the systems jurisdiction for example writing into a socket then the *disclosure policy* code is executed. System calls who send send data outside of the system can pose privacy leaks thus the *jurisdiction policy* defines how the user wishes to handle such situations.

7.2 Formal Languages

One challenge that appears more prominent in dynamic analysis than in static is handling *implicit flows*, when code whose execution is conditional to private information updates a public variable. Austin and Flanagan [1] [2] have study the problem for dynamic languages. In the beginning they propose the *No-Sensitive-Upgrade Check*, where they prohibit the update of public variables from a confidential program counter. In their second paper they propose the *permissive-upgrade* strategy. Information leaks caused by implicit flows are tracked using a special label.

There are multiple static and dynamic systems for controlling information flow. SELinks [6] is a security-enhanced version of the Links web-programming language, extended with support for typed labels. SELinks supports persistent labels through the database, since all client, server, and database code is generated by the Links compiler from the same SELinks web-program. Jif [19, 17] is an extension of Java with support for label-based information flow. It uses a combination of type-checking [37], static analysis and runtime checks to enforce information-flow policies in Jif programs. Banerjee and Naumann [3] present a similar static type-checking system for statically checking label-based policies in object oriented languages. Functional languages like Fable, Fine and F* [28, 27, 26] support complex, dependent label types that are capable of expressing and enforcing complicated policies, dynamic label creation.

Taint analysis is an important sub-problem of information flow, and has been studied extensively in the past. Static taint analysis [10, 15] for C and Java use type-based static analysis to infer tainting for all possible static labels in the program, providing sound guarantees, although they suffer from false positives. Dynamic taint analysis for Perl [32] and Java [11] change the interpreter or VM to track tainting information per unit of data, either per character or per object. Php-Taint [30] extends the PHP engine with similar per-object support, although it is not fully maintained in the current PHP engine. In LabelFlow, we extended PHP-Taint with support for arbitrary labels, external C library functions and the PHP foreign function interface, as well as more language primitives. Many systems have been proposed in the past for controlling information flow in the database. LabelFlow supports row-level label granularity, similarly to *row-level security* supported by several commercial relational databases. Li and Zdancewic [14] present a label-based formal system for checking information flow through the database in web applications and prove its safety.

Chapter 8

Conclusions

Web applications are highly complex and sophisticated, usually composed of many diverse components and layers, and often written in different languages. This makes it hard for the programmer to change an existing web application to control information flow or adhere to a specific privacy policy. This paper presents LabelFlow, a system for dynamic information flow tracking on web applications in PHP. LabelFlow improves security and privacy in legacy web applications using label-based information flow. LabelFlow *handles* the multi-tier architecture usually found in web applications; it *transparently* extends the database schema to associate information flow labels with every row; it extends the PHP bytecode interpreter to transparently track labels at runtime; and it combines the two, so that the programmer need only implement the policy code with *minimal, or even zero,* changes to the rest of the legacy application.

We evaluated LabelFlow on three large real-world web applications. With minimal code changes, LabelFlow was able to enforce complex policies with minimal overhead. Finally, we have formally proven that our extensions protect against information leakage.

Bibliography

[1] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.

[2] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

[3] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *CSFW*, 2002.

[4] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security*, pages 1–1, 2003.

[5] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *SOSP*, SP '10, pages 191–206, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD*, July 2009.

[7] Benjamin Davis and Hao Chen. Dbtaint: Cross-application information flow tracking via databases. In *WebApps*, 2010.

[8] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5):17–30, 2005.

[9] Federal Trade Commission. Facebook settles ftc charges that it deceived consumers by failing to keep privacy promises. `http://www.ftc.gov/opa/2011/11/privacysettlement.shtm`, November 2011.

[10] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-Insensitive Type Qualifiers. *TOPLAS*, 28(6):1035–1087, November 2006.

[11] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *ACSAC*, 2005.

[12] C. Kambalyal. 3-tier architecture. `http://channukambalyal.tripod.com/NTierArchitecture.pdf`, 2010.

[13] L.A Times. Bank of america data leak destroys trust. `http://articles.latimes.com/2011/may/24/business/la-fi-lazarus-20110524`, May 2011.

[14] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *CSFW*, 2005.

[15] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security*, 2005.

[16] MediaWiki.org. Security issues with authorization extensions. `http://www.mediawiki.org/wiki/Security_issues_with_authorization_extensions`, August 2011.

[17] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL*, 1999.

[18] A. C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ToSEM*, 9:2000, 2000.

[19] A. C. Myers, N. Nystrom, L. Zheng, , and S. Zdancewic. Jif: Java information flow. `http://www.cs.cornell.edu/jif`, July 2001. Software Release.

[20] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS*, 2009.

[21] S. Nanda, L.C. Lam, and T. Chiueh. Dynamic Multi-Process Information Flow Tracking for Web Application Security. In *Middleware*, 2007.

[22] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[23] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *IFIP SEC*, pages 372–382, 2005.

[24] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *NDSS*, San Diego, CA, February 8-11, 2009.

[25] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: taint analysis of framework-based web applications. *SIGPLAN Not.*, 46:1053–1068, October 2011.

[26] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *POPL*, 2012.

[27] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in fine. In *ESOP*, 2010.

[28] Nikhil Swamy, Brian Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *SOSP*, 2008.

[29] The Internet Society. Hypertext transfer protocol – http/1.1, 1990.

[30] Wietse Venema. Taint support for PHP, April 2011. `https://wiki.php.net/rfc/taint`. Last visited on January 2012.

[31] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[32] L. Wall, T. Christiansen, and J. Orwant. *Prog. Perl.* O'Reilly, 3 edition, 2000.

[33] WordPress.org. Warning for context visibility, 2012.

[34] Wei Xu, Eep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX Security*, pages 121–136, 2006.

[35] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.

[36] A.R. Yumerefendi, B. Mickle, and L.P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, pages 159–172, 2007.

[37] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *ESOP*, 2001.

[38] K. Zhang, Z. Li, R. Wang, X.F. Wang, and S. Chen. Sidebuster: automated detection and quantification of side-channel leaks in web application development. In *CCS*, pages 595–606, 2010.

Appendix A

Formal Semantics

A.1 Language Terms

$$
\begin{array}{lrcl}
\text{(Constants)} & n & \in & \mathbb{N} \\
\text{(Base Labels)} & k & \in & \mathcal{L} \\
\text{(Labels)} & l, pc & ::= & k \mid x \mid l \sqcup l \mid \bot \mid \top \\
\text{(Constraints)} & C & ::= & \emptyset \mid C, l \sqsubseteq l \\
\text{(Values)} & v & ::= & l \mid n^l \mid () \mid \lambda x \,.\, e \\
\text{(Expressions)} & e & ::= & v \mid e\ e \mid \textsf{create table} \mid \textsf{insert}\ e\ \textsf{into}\ n \\
 & & \mid & \textsf{update}\ e\ \textsf{to}\ e\ \textsf{in}\ n \mid \textsf{newlabel} \\
 & & \mid & \textsf{taint}\ e\ \textsf{with}\ e \mid \textsf{elevate}\ e_p \\
\text{(Databases)} & DB & ::= & \emptyset \mid DB, T \\
\text{(DB Tables)} & T & \subseteq & \emptyset \mid T, (n, l)
\end{array}
$$

A.1.1 Operational Semantics

$$\text{E-New}\ \frac{l - \text{fresh}}{\langle DB, pc, \mathsf{newlabel}\rangle \rightarrow \langle DB, pc, l\rangle}$$

$$\text{E-Create}\ \frac{}{\langle DB, pc, \mathsf{create\ table}\rangle \rightarrow \langle (DB, \emptyset), pc, ()\rangle}$$

$$\text{E-Insert}\ \frac{T'_k = T_k, (v, pc)}{\begin{array}{r}\left\langle T_1, \ldots, T_k, \ldots T_n, pc, \mathsf{insert}\ v^l\ \mathsf{into}\ k\right\rangle \rightarrow \\ \langle T_1, \ldots, T'_k, \ldots T_n, pc, ()\rangle\end{array}}$$

$$\text{E-Select}\ \frac{T_n \in DB \qquad (v, l_2) \in \{(v, l) \mid (v, l) \in T_n \wedge l \sqsubseteq pc\}}{\left\langle DB, pc, \mathsf{select}\ v^{l_1}\ \mathsf{from}\ n\right\rangle \rightarrow \left\langle DB, pc, v^{l_2}\right\rangle}$$

$$\text{E-Taint1}\ \frac{e_1 \rightarrow e'_1}{\langle DB_1, pc, \mathsf{taint}\ e_1\ \mathsf{with}\ e_2\rangle \rightarrow \langle DB_1, pc, \mathsf{taint}\ e'_1\ \mathsf{with}\ e_2\rangle}$$

$$\text{E-Taint2}\ \frac{e \rightarrow e}{\langle DB, pc, \mathsf{taint}\ v\ \mathsf{with}\ e\rangle \rightarrow \langle DB, pc, \mathsf{taint}\ v\ \mathsf{with}\ e'\rangle}$$

$$\text{E-Taint3}\ \frac{l = labelof(n) \qquad lab(n, l \sqcup pc)}{\langle DB, pc, \mathsf{taint}\ n\ \mathsf{with}\ l\rangle \rightarrow \langle DB, pc, l\rangle}$$

$$\text{E-Elevate1}\ \frac{\langle DB, \top, \mathsf{elevate}\ e\rangle \rightarrow \langle DB, \top, \mathsf{elevate}\ e'\rangle}{\langle DB, pc, \mathsf{elevate}\ e\rangle \rightarrow \langle DB, pc, \mathsf{elevate}\ e'\rangle}$$

$$\text{E-Elevate2}\ \frac{}{\langle DB, pc, \mathsf{elevate}\ l\rangle \rightarrow \langle DB, l, ()\rangle}$$

$$\text{E-App}\ \frac{}{(\lambda x.e)v \rightarrow e[v/x]}$$

$$\text{E-App1}\ \frac{e_1 \rightarrow e'_1}{e_1\, e_2 \rightarrow e'_1\, e_2}$$

$$\text{E-App2}\ \frac{e_2 \rightarrow e'_2}{v_1\, e_2 \rightarrow v_1\, e'_2}$$

A.2 The Language

PHP_{sec} is a functional language designed to resemble PHP and our changes for information flow. Our language includes a database for persistent storage.

A.2.1 Semantics

New Create a new label.

Create Create a new table. From the programmer's perspective each table has only one column, thus each row contains only one piece of data. Internally, our engine augments the table with one extra column to hold the label for each piece of data is stored in the table.

Select Select the elements from the table with label no more restrictive than the current *pc*.

Insert Insert a piece of data into the table. Internally, the label of the data is stored in a hidden column in the same row as the data.

Taint Taint a piece of data with the label. The new label is appended to the labels that the data already have.

Elevate Elevate upgrades the *pc* of the current session.Elevate executes an application specific code that authenticates the user with the application and returns its credential. The credential is the most restrictive label that the user can access. Elevate upgrades the *pc* to $\top$, so that the authentication code can have full access to all data. In the end Elevate updates the current *pc* to the credential of the user.

A.3 Formal Proofs

Lemma 1. *For each label l generated by newlabel the following conditions always hold:*

1. $\bot \sqsubset l \sqsubset \top$
2. $l \sqsubseteq l', \forall l' \in \mathcal{L} - \{\bot\}$

Lemma 2 (Table Similarity)**.** *Let tables $T_1, T_2 \subseteq \mathbb{N} \times \mathcal{L}$. We say that T_1 and T_2 are similar up to l and write $(T_1 =_l T_2)$, if $\forall l' \sqsubseteq l, v\ (v, l') \in T_1 \Leftrightarrow (v, l') \in T_2$.*

The only difference between T_1 and T_2 is for data with label more restrictive than l.

Lemma 3. *Let databases $DB_1 = \{T_1, \ldots, T_n\}$ and $DB_2 = \{T'_1, \ldots, T'_k\}$. The two databases have the same schema iff $n = k$.*

Since each table consists of only one column and the is only a single type of data. Two databases have the same schema if they contain the same number of tables.

Lemma 4 (Database Similarity)**.** *Let databases $DB_1 = \{T_1, \dots, T_n\}$ and $DB_2 = \{T'_1, \dots, T'_k\}$. We say that DB_1 and DB_2 are similar up to l and write $DB_1 =_l DB_2$ if:*

1. *$n = k$, thus the databases have the same schema.*
2. *$\forall i \in (1, n), T_i \in DB_1, T'_i \in DB_2 \Rightarrow T_i =_l T'_i$*

Two databases are similar up to l if the have the same schema and each table T_i in DB_1 is similar up to l with T'_i in DB_2.

Theorem 2. *Assume e is an expression without any $\mathsf{elevate}\ e$ terms, l and pc are labels, and DB_1, DB_2 are databases with $DB_1 \sim_l DB_2$. Then executing e under the two different databases with input labeled l will yield the same results: $\langle DB_1, pc, e\rangle \to^* \langle DB'_1, pc, v\rangle$ if and only if $\langle DB_2, pc, e\rangle \to^* \langle DB'_2, pc, v\rangle$ Moreover, it will be $DB'_1 \sim_l DB'_2$.*

Proof. By induction, on the derivations of e. We will show that executing each derivation of e, with pc and label l for eachd database will return the same result.

Base Cases For following cases the theorem holds trivially:

value v

label l

unit ()

function $\lambda x.e$

Induction Assuming that the theorem holds for an expression e we are going to prove that it holds for e'.

newlabel Assume $e = \mathsf{newlabel}$ then the rule [E-NEW] applies:

$$\langle DB_1, pc, \mathsf{newlabel}\rangle \to \langle DB_1, pc, l\rangle \tag{A.1}$$

$$\langle DB_2, pc, \mathsf{newlabel}\rangle \to \langle DB_2, pc, l\rangle \tag{A.2}$$

Both (1) and (2) return the same result.

taint if $e = \mathsf{taint}\ e_1\ \mathsf{with}\ e_2$. There are three cases to consider:

1. If e_1 can step then rule [E-Taint1] applies:

$$\langle DB_1, pc, \mathsf{taint}\ e_1\ \mathsf{with}\ e_2\rangle \to \left\langle DB_1, pc, \mathsf{taint}\ e'_1\ \mathsf{with}\ e_2\right\rangle \tag{A.3}$$

$$\langle DB_2, pc, \mathsf{taint}\ e_1\ \mathsf{with}\ e_2\rangle \to \left\langle DB_2, pc, \mathsf{taint}\ e'_1\ \mathsf{with}\ e_2\right\rangle \tag{A.4}$$

2. If e_2 can step then rule [E-Taint2] applies.

$$\langle DB_1, pc, \textsf{taint } v \textsf{ with } e_2\rangle \rightarrow \left\langle DB_1, pc, \textsf{taint } v \textsf{ with } e_2'\right\rangle \tag{A.5}$$

$$\langle DB_2, pc, \textsf{taint } v \textsf{ with } e_2\rangle \rightarrow \left\langle DB_2, pc, \textsf{taint } v \textsf{ with } e_2'\right\rangle \tag{A.6}$$

3. If $e_1 = v$ and $e_2 = l$ then by [E-Taint3].

$$\langle DB_1, pc, \textsf{taint } v \textsf{ with } l\rangle \rightarrow \langle DB_1, pc, l\rangle \tag{A.7}$$

$$\langle DB_2, pc, \textsf{taint } v \textsf{ with } l\rangle \rightarrow \langle DB_2, pc, l\rangle \tag{A.8}$$

create Assume $e = \textsf{create table}$ then rule [E-Create] applies:

$$\langle DB_1, pc, \textsf{create table}\rangle \rightarrow \langle DB_1, \emptyset, pc, ()\rangle \tag{A.9}$$

$$\langle DB_2, pc, \textsf{create table}\rangle \rightarrow \langle (DB_2, \emptyset), pc, ()\rangle \tag{A.10}$$

insert Let $DB_1 = \{T_{11}, \ldots, T_{1n}\}$ and $DB_2 = \{T_{21}, \ldots, T_{2n}\}$. If $e =$ *insert* v^l *into* k and e can take a step then the rule [E-INSERT] applies:

$$\left\langle \{T_{11}, \ldots, T_{1k}, \ldots, T_{1n}\}, pc, \textsf{insert } v^l \textsf{ into } k\right\rangle \rightarrow \left\langle \{T_{11}, \ldots, T_{1k}', \ldots, T_{1n}\}, pc, ()\right\rangle \tag{A.11}$$

$$\left\langle \{T_{21}, \ldots, T_{2k}, \ldots, T_{2n}\}, pc, \textsf{insert } v^l \textsf{ into } k\right\rangle \rightarrow \left\langle \{T_{21}, \ldots, T_{2k}', \ldots, T_{2n}\}, pc, ()\right\rangle \tag{A.12}$$

We need to prove that $DB_1 =_l DB_2$ still holds after the insert. We start by proving that $T_{1k}' =_l T_{2k}'$.

$$T_{1k}' = T_{1k} \cup \{(v, l)\} \tag{A.13}$$

$$T_{2k}' = T_{2k} \cup \{(v, l)\} \tag{A.14}$$

Since $T_{1k} =_l T_{2k}$ holds follows that $T_{1k}' =_l T_{2k}'$. By lemma 4 follows that $DB_1 =_l DB_2$ holds.

select If $e = \textsf{select } k \textsf{ from } n$ the rule [E-Select] applies:

$$\langle DB_1, pc, \textsf{select } k \textsf{ from } n\rangle \rightarrow \langle DB_1, pc, v_{1k}\rangle \tag{A.15}$$

$$\langle DB_2, pc, \textsf{select } k \textsf{ from } n\rangle \rightarrow \langle DB_2, pc, v_{2k}\rangle \tag{A.16}$$

We will show that $v_{1k} = v_{2k}$. By the premises of the [E-Select] inference rule we know that:

$$v_{1k} = \{(v,l) \mid (v,l) \in n_1 \wedge l \sqsubseteq pc\} \tag{A.17}$$

$$v_{2k} = \{(v,l) \mid (v,l) \in n_2 \wedge l \sqsubseteq pc\} \tag{A.18}$$

Since table n in DB_1 and table n in DB_2 contain the same elements up to pc and v_{ik} will only retrive elements up to pc folows that $v_1k = v_2k$.

e · e If $e = e_1 \cdot e_2$, we have three subcases to consider.

1. If e_1 can make an evaluation step then the rule [E-App1] applies:

$$\langle DB_1, pc, e_1 \cdot e_2\rangle \to \left\langle DB_1, pc, e_1' \cdot e_2\rangle\right\rangle \tag{A.19}$$

$$\langle DB_2, pc, e_1 \cdot e_2\rangle \to \left\langle DB_2, pc, e_1' \cdot e_2\rangle\right\rangle \tag{A.20}$$

2. If e_1 is a value and e_2 can make an evaluation step then the rule [E-App2] applies:

$$\langle DB_1, pc, e_1 \cdot e_2\rangle \to \left\langle DB_1, pc, e_1 \cdot e_2'\right\rangle \tag{A.21}$$

$$\langle DB_2, pc, e_1 \cdot e_2\rangle \to \left\langle DB_2, pc, e_1' \cdot e_2\right\rangle \tag{A.22}$$

3. If $e_1 = \lambda x.e$ is a value and $e_2 = v$ is a value then the rule [E-APP] applies:

$$\langle DB_1, pc, (\lambda x.e)v\rangle \to \langle DB_1, pc, e[v/x]\rangle \tag{A.23}$$

$$\langle DB_2, pc, (\lambda x.e)v\rangle \to \langle DB_2, pc, e[v/x]\rangle \tag{A.24}$$

□