Practical Information Flow for Legacy Web Applications

Georgios Chinis
Foundation of Research and Technology—Hellas

Polyvios Pratikakis
Foundation of Research and Technology—Hellas

Elias Athanasopoulos
Columbia University, NY

Sotiris Ioannidis
Foundation of Research and Technology—Hellas

ABSTRACT

The popularity of web applications, coupled with the data they operate on, makes them prime targets for hackers 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 paper 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. We used LabelFlow to add and enforce access control policies in three popular real-world large scale web applications: MediaWiki, Wordpress and OpenCart. LabelFlow requires minimal code changes of 50–100 lines of code per application, while incurring little execution overhead of up to 5.6% at worst.

1. INTRODUCTION

Controlling the flow of information is paramount to the security of applications. Ensuring data confidentiality, integrity and enforcing security policies in an application, all rely on managing and restricting access to data and their flow. 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 have 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, for example, 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 [27] 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: 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 3% 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, requiring minimum code changes. More precisely, we applied LabelFlow on MediaWiki, the software that runs Wikipedia, using less than 100 lines of code; on WordPress, a popular blogging tool, in 60 lines of code; and on OpenCart, an online-store management application, in less than 60 lines of code.

The rest of this paper is organized as follows. Section 2 provides background information on information flow policy mechanisms. Section 3 presents the architecture of LabelFlow and discusses the functionality of each component. Section 5 discusses the LabelFlow implementation. Section 6 reports on qualitative and quantitative measurements of the performance and usability of LabelFlow. Section 7 covers related work and we conclude in Section 8.

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 what 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 at 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.

Side channels, like time attacks [4, 38, 3], the program's execution flow, power analysis, etc., can also cause information leaks. To protect against such leaks, a secure information

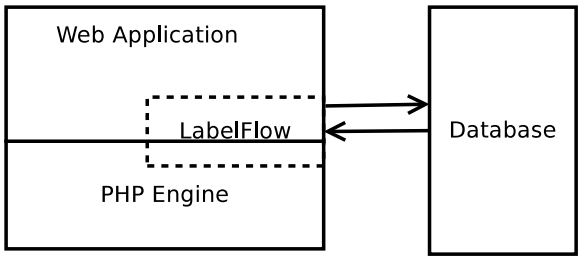


Figure 1: The architecture of LabelFlow

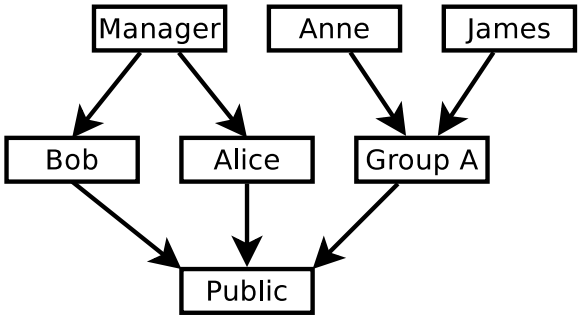


Figure 2: Label graph: A semi-lattice representing the relation between the labels

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.

3. 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.

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. Overall, our system is used as follows.

3.1 Application Layer

Initially, the programmer must label sensitive data that need to be monitored using our API. Deciding which data need labeling depends on privacy policy the developer wishes to enforce. For instance, if the developer wishes 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. 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). Section 5 discusses propagation in detail.

3.2 Storage Layer

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. This conservative policy can restrict the label of some fields even fur-

ther, when, for example, many public data items are stored in the same tuple with a secret data item. This conservative policy may elevate the label of some data but protects against data leaks.

3.3 Label Graph

Consider the secure MediaWiki application example described in Section 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 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.

4. 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 an accompanying technical report [5].

Figure 3 presents a simple functional language with support for dynamic labels and abstract 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

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

Figure 3: A simple calculus with dynamic labels and persistent state

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 Virtual Machine. All constants in the program code are trivially annotated with the label $\bot$.

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 k inserts the result of expression e into the k-th table of the database; and expression select e from n looks up the result of e in the n-th table of the database. Moreover, expression newlabel creates and returns a new label at run time; expression taint e_1 with e_2 computes e_1 to an integer, e_2 to a label, and taints the integer with the new label; and expression elevate e_p computes pure expression e_p (which should not have side effects in the database) to a label, and sets the current elevation state to that label.

4.1 Operational Semantics

Figure 4 presents the small-step operational semantics for the language. Judgments have the form

$$\langle DB, pc, e\rangle \rightarrow \langle DB', pc', e'\rangle$$

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-App] is standard function application.

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 n^l into the database, using both the current elevation level and the value's label $pc \sqcup l$. 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.

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-Taint] updates the label of a value n to the given label l' also tainting it with the current elevation level pc. Rules [E-Elevate-

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

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

$$\text{[E-Insert]}\frac{T_k' = T_k, (n, l \sqcup pc)}{\langle T_1, \ldots, T_k, \ldots T_n, pc, \mathsf{insert}\ n^l\ \mathsf{into}\ k\rangle \to \langle T_1, \ldots, T_k', \ldots T_n, pc, ()\rangle}$$

$$\text{[E-Select]}\frac{T_k \in DB \quad (n, l_2) \in \{(m, l) \mid (m, l) \in T_k \wedge l \sqsubseteq pc\}}{\langle DB, pc, \mathsf{select}\ n^{l_1}\ \mathsf{from}\ k\rangle \to \langle DB, pc, n^{l_2}\rangle}$$

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

$$\text{[E-Taint]}\frac{}{\langle DB, pc, \mathsf{taint}\ n^l\ \mathsf{with}\ l'\rangle \to \langle DB, pc, n^{l' \sqcup pc}\rangle}$$

$$\text{E-Elevate-1}\frac{\langle DB, \top, \mathsf{elevate}\ e\rangle \to \langle DB, \top, \mathsf{elevate}\ e'\rangle}{\langle DB, pc, \mathsf{elevate}\ e\rangle \to \langle DB, pc, \mathsf{elevate}\ e'\rangle}$$

$$\text{E-Elevate-2}\frac{}{\langle DB, pc, \mathsf{elevate}\ l\rangle \to \langle DB, l, ()\rangle}$$

Figure 4: Operational semantics rules

1] and [E-Elevate-2] execute policy code. Namely, to execute expression elevate e we require in [E-Elevate-1] that e does not change the database DB. The policy code e can only compute and return a label l which is set to be the new elevation label at the end of the elevate e expression in [E-Elevate-2].

4.2 Soundness

We use the semantics to prove that any code not using elevate e instructions satisfies noninterference, i.e., cannot leak any data labeled by a label above its pc. To do that, we introduce the following definitions.

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*

$$\forall l' \sqsubseteq l, v\ (v, l') \in T_1 \Leftrightarrow (v, l') \in T_2$$

Intuitively, two tables are similar up to label l if the only differences between T_1 and T_2 occur on data with label more restrictive than l.

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*

$$\forall 1 \leq i \leq n, T_i \in DB_1, T_i' \in DB_2 \Rightarrow T_i \sim_l T_i'$$

Intuitively, 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, meaning the two databases only differ on data labeled with a label more restrictive than l.

Theorem 1. *Assume that e is an expression without any* 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 $DB_1' \sim_l DB_2'$.

We prove this bisimilarity theorem by straightforward induction. Details of the proof can be found in an accompanying technical report [5].

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 includes 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, potentially leading to 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.

Label representation. 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. We extended the internal representation of userspace variable to carry the label.

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 currently support up to 32

```
1  CREATE TABLE ( fname   VARCHAR(100),
2                 lname   VARCHAR(100),
3                 address VARCHAR(255))
4
5
6  INSERT INTO
7    table_name (fname, lname, address)
8    VALUES (1, 2, 3)
9
10 SELECT fname, lname, address
11   FROM table_name
12   WHERE condition
```

(a) Original SQL code

```
1  CREATE TABLE ( fname   VARCHAR(100),
2                 lname   VARCHAR(100),
3                 address VARCHAR(255),
4                 label_ac SET(...) default 1)
5
6  INSERT INTO
7    table_name (fname, lname, address, label_ac)
8    VALUES (1, 2, 3, label)
9
10 SELECT fname, lname, address, label_ac
11   FROM table_name
12   WHERE ((label_ac | user_label)=user_label)
13         AND (condition)
```

(b) SQL code after rewriting

Figure 5: Example SQL queries, rewritten by LabelFlow.

labels. The number of labels is limited but easy to extend with minimal cost. Additionally, one-hot encoding of the label permits very fast manipulation of the taint bit using bitwise operations, making label union and "meet" quite fast.

LabelFlow propagates 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.

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 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, normally a relational database, where they reliably store information essential for their normal operation. 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 label ware 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.

LabelFlow performs the necessary modifications of the database schema and any queries inserting and retrieving data from the database, 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 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 shows a representative example of SQL rewrites. The first query shown in Figure 5(a) (lines 1–3) originally creates a table with three fields. LabelFlow intercepts the query and rewrites it as shown in Figure 5(b). The `CREATE TABLE` query is rewritten to include an extra field to store the label for each row, shown in Figure 5(b) (line 4). The second query shown in (a), lines 6–8, inserts a tuple in the table. LabelFlow rewrites this to also insert a value in the label field, shown in (b), lines 6–8. The label value corresponds to the union of all fields' labels. Finally, the third query shown in (a), lines 10–12 performs a selection on the table. We rewrite this to also constrain the row label to the label of the user performing the query, shown in (b),

lines 10–13. 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 register meaningful names as labels, (ii) an API for constructing the label graph discussed in Section 3.3, 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.

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 stores the mapping between the application-level representation of the labels and the low-level integer representation in a new table in the database. Finally, LabelFlow uses a second table to hold the label graph.

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

1. Incorporate LabelFlow with the application by including LabelFlow in the main file of the application and instantiating the LabelFlow object. The LabelFlow object constructor requires the credentials to a database for storing its internal tables.

2. Define the principals and the label graph. Each application defines different kinds of principals. For instance in MediaWiki the principals are users, in OpenCart they are users and customers. The label graph represents the relation between the principals. We assume that the application provides mechanisms for creating and authenticating principals.

3. Extend the existing mechanism that creates principals to associate a label with each new principal and insert code to taint all data of the new principal with that label.

4. Extend the existing authentication mechanism to elevate the *pc*. Note that in the beginning of a web PHP request the *pc* is *public*, while all users' data in the database are tainted with their respective label. So, the authentication code will normally not be able to retrieve any user information. To solve this problem the programmer must call the authentication code using *elevate*. Inside *elevate* the *pc* has a special value *system*, from which all data are accessible and authentication is possible. This would be unsound in general, we require any policy code used for authentication to not have side effects. Namely, inside elevate the application cannot perform state modifying queries like `INSERT` or `UPDATE`; only `SELECT`. To defend against malicious code injections that may call `elevate` to gain unlimited access data, `elevate` cannot be called inside `eval` (or any equivalent, using e.g., `call_user_func`).

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.

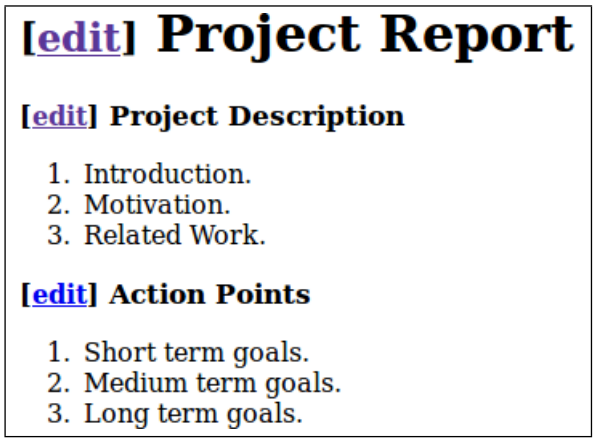
To test LabelFlow's applicability and ease of use, we used three widely used applications: MediaWiki, the wiki used by Wikipedia; WordPress, a blog hosting application; and OpenCart, an e-commerce and store management application. We run all experiments on a Pentium 4, 3.4GHz workstation with 3 GB of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

6.1 MediaWiki

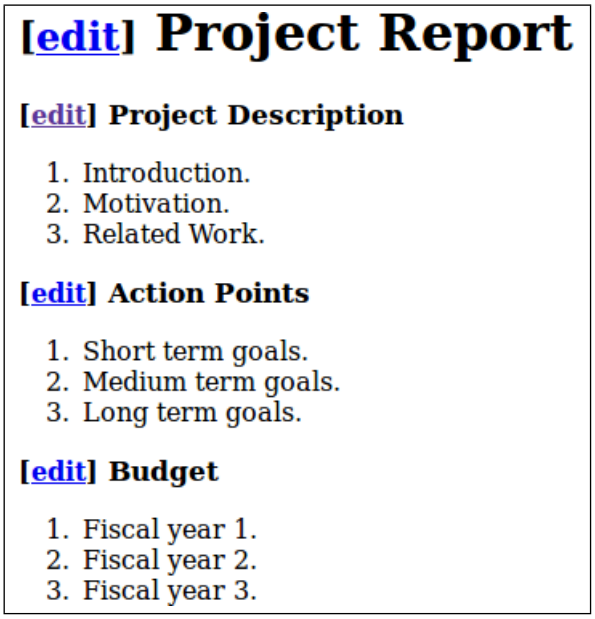
In MediaWiki, users modify the articles and create new revisions. Using LabelFlow we implemented an access control policy where each user that creates a revision labels it with his credential. Other users who wish to read the article have access only to the revisions accessible from their credential.

For instance, Figures 6(a) and 6(b) show an article as viewed by two different users. The article is a progress report about a project. The first user 6(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]. For some, MediaWiki offers suggestions on how to overcome them. We focused on the ones that offer no such suggestions (see Table 1). To the best of our knowledge LabelFlow is able to offer protection against all of these vulnerabilities.



(a) View of project member



(b) View of project manager

Figure 6: The same page of our wiki as seen by two different users with different authorizations.

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 1: Common Vulnerabilities

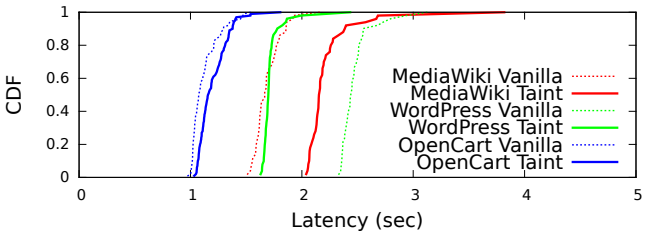
The necessary modifications to enforce the policy on MediaWiki were fairly straightforward, totaling around 100 lines of additional code in a code base of over 100,000 lines. Moreover, they were often made apparent by helpful error messages while migrating to LabelFlow, when MediaWiki encountered an 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 7 (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.

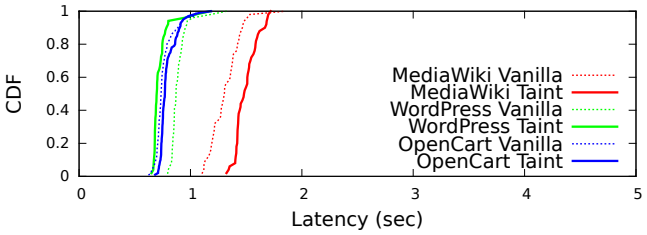
Figure 7 (b), shows the total time needed to retrieve an article from the database. MediaWiki must retrieve the user's information based on their cookie, find the appropriate revision for the particular article for the user and finally retrieve the text. In both experiments, LabelFlow imposes only a small overhead, because of its efficient label representation and fast query rewriting.

6.2 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:



(a) User login.



(b) Full page load.

Figure 7: Cumulative Distribution Function (CDF) of the time for two kinds of requests.

> *"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]

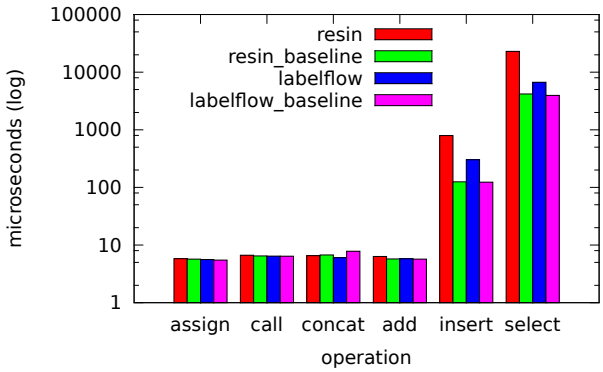
In WordPress, users do not create accounts for themselves, they instead rely on the administrator to create the accounts for them. Thus, initially the administrator must have access to user data, but should drop it as soon as possible. We encoded this behavior by having the administrator code create 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.

6.3 OpenCart

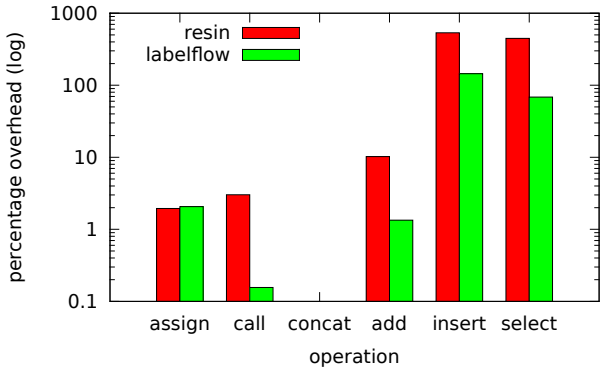
OpenCart is an e-commerce and online store-management software program. In OpenCart, system administrators add products available for purchase, and customers place orders and write reviews about the products they have purchased. OpenCart follows the MVC model, where the code is divided into three categories: Model, View and Controller. Model is the database abstraction layer, View is responsible for the presentation of the information and Controller implements the application logic. Despite the difference in the architecture, 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.

6.4 Comparison with Resin

Resin [35] is an information-flow system for PHP that uses assertion-based data flow. Assertions are pieces of code that



(a) Microbenchmark performance.



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

Figure 8: Comparison between LabelFlow and Resin.

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 7). 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 8 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.

7. RELATED WORK

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]; a language runtime that supports dynamically checking assertions in PHP and Java programs. Resin requires the programmer to write policy assertions and modifies the PHP runtime engine to

dynamically check and enforce the described policies. To do that, it performs dynamic tracking of application data, similar to information flow tracking in LabelFlow. LabelFlow provides an application agnostic representation of the policy which we believe is easier to implement in legacy systems. Measurements have shown that LabelFlow adds a smaller overhead to the application than Resin. DBtaint [7] adds information flow tracking in the Perl and Java database API. Similarly to LabelFlow, DBTaint replaces each piece of data in the database with a composite representation of the data and its taint value. It then dynamically rewrites queries to extract the taint bit or data value as required. LabelFlow also uses dynamic SQL rewriting to insert labels into the database. It, however, labels the whole row in a table on `INSERT` and `UPDATE` queries, whereas it ignores rows with higher labels on `SELECT`. To facilitate porting legacy code, we also perform dynamic rewriting of `CREATE TABLE` queries, so that all changes in the schema are transparent and no database code needs to be rewritten in the application.

The research community has identified intrinsic problems in taint analysis [25], nevertheless, there is active research towards more efficient and faster frameworks [2]. More particularly, tainting has been extensively used in various proposals for securing a wide range of systems. Newsome et al., have proposed dynamic tainting analysis for detecting exploits on commodity systems [22], privacy leakage in the cloud environment [36] and in smartphones [8]. Tainting has been also used solely in securing web applications [31, 21, 23], and, partially, for detecting and preventing code-injection attacks [20, 24]. 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.

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 [1] present a similar static type-checking system for statically checking label-based policies in object oriented languages. Functional languages like Fable, Fine and F* [29, 28, 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[1] [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.

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.

9. REFERENCES

[1] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE workshop on Computer Security Foundations*, 2002.

[2] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Proceedings of RAID'11*, Menlo Park, CA, September 2011.

[3] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 1–1, 2003.

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

[1] `http://perldoc.perl.org/perlsec.html#Taint-mode`

[5] Georgios Chinis, Polyvios Pratikakis, Elias Athanosopoulos, and Sotiris Ioannidis. Practical information flow for legacy web applications. Technical Report 428-Apr-2012, Foundation for Research and Technology - Hellas, April 2012.

[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 *Proceedings of the 2010 USENIX conference on Web application development*, 2010.

[8] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[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. *ACM Transactions on Programming Languages and Systems*, 28(6):1035–1087, November 2006.

[11] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 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 *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, 2005.

[15] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, 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 *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999.

[18] A. C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, October 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 *In Proceedings of the Network and Distributed System Security Symposium*, 2009.

[21] S. Nanda, L.C. Lam, and T. Chiueh. Dynamic Multi-Process Information Flow Tracking for Web Application Security. In *Proceedings of the 2007 ACM/IFIP/USENIX international conference on Middleware companion*, 2007.

[22] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *In Proceedings of the Network and Distributed System Security Symposium*, 2005.

[23] Anh Nguyen-tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[24] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *In Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 8-11, 2009.

[25] Asia Slowinska and Herbert Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of ACM SIGOPS EUROSYS*, Nuremberg, Germany, March-April 2009.

[26] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.

[27] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems.*, 2004.

[28] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in fine. In *Proceedings of the 19th European conference on Programming Languages and Systems*, 2010.

[29] Nikhil Swamy, Brian Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[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 *In Proceedings of the Network and Distributed System Security Symposium*, 2007.

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

[33] Wordpress.org. Content visibility. `http://codex.wordpress.org/Content_Visibility`, August 2011.

[34] Wei Xu, Eep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, pages 121–136, 2006.

[35] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 291–304, 2009.

[36] 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.

[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 *Proceedings of the 17th ACM conference on Computer and communications security*, pages 595–606, 2010.