Practical Information-flow Control in Web-based Information Systems

Peng Li Steve Zdancewic
University of Pennsylvania

Abstract

This paper presents a practical application of language-based information-flow control, namely, a domain-specific web scripting language designed for interfacing with databases. The primary goal is to provide strong enforcement of confidentiality and integrity policies: Confidential data can be released only in permitted ways, and trustworthy data must result from expected computations or conform to expected patterns. Such security policies are specified in the database layer and statically enforced for the rest of the system in an end-to-end fashion.

In contrast with existing web-scripting languages, which provide only ad hoc mechanisms for information security, the scripting language described here uses principles based on the well-studied techniques in information-flow type systems. However, because web scripts often need to downgrade confidential data and manipulated untrusted user input, they require practical and convenient ways of downgrading secure data. To achieve this goal, the language allows safe downgrading according to *downgrading policies* specified by the programmer. This novel, pattern-based approach provides a practical instance of recent work on *delimited release* and *relaxed noninterference* and extends that work by accounting for integrity policies.

1 Introduction

This paper presents a language-based approach to enforcing confidentiality and integrity of data in typical web-based information systems. Such systems are usually implemented using a layered design in which data is stored in the database and accessed by using a web browser. The database management system (DBMS) provides a data query language (for example, SQL) to store, modify and extract information from databases. Application software connects to the DBMS via some programming language interface and submits *queries* as requests for information from the DBMS. The application software then processes the data and send the computation results to the end-users, typically bundled as HTML. This paper focuses on a simple yet widely used design, where relational databases such as MySQL are used as DBMSs and web scripting languages such as PHP are used for developing application software.

In practice, there are many security concerns for such systems. For example, an unsafe PHP script could use strings from untrusted inputs to compose SQL queries and then have the DBMS execute the query, which potentially allows an attacker to insert arbitrary commands in the SQL query. PHP scripts that access confidential data in the database must also release them only in permitted ways. For example, one might require that a password can be compared against user inputs but cannot be printed to the web page verbatim, or that only the last several digits of a social security number or credit card number should be displayed in the HTML output.

As these web applications become complex, the security of the system becomes hard to manage. In the worst case, the programmer must walk through all the code and check every line to make sure that there are no security violations. In addition, the queries performed by the script must comply with the desired policies on the data stored in the database. Ensuring that all of the security requirements are met is difficult to do manually, tedious, and error-prone.

Currently, there exist only ad hoc ways to (partially) enforce such security policies. To prevent accidental use of untrusted inputs as parameters to safety-critical operations—thereby preventing format string attacks and malicious DBMS queries—some scripting languages provide mechanisms to track the uses of untrusted inputs, dynamically checking that they are not used inappropriately. Perl, for instance, can be run in a "taint checking" mode in which user input strings are considered tainted until they are matched against a programmer-supplied pattern, which establishes that the untrusted input actually conforms to an expected form.

In an effort to enforce the confidentiality policies on the database, experienced software developers implement most of the security-sensitive operations in the DBMS as procedure calls in the query language, exposing explicit, restricted interfaces. Programmers should use only these interfaces to perform queries in the web scripting language. This limited interface means that the database query engine can itself enforce the desired policies. Such encapsulation means that the programmer needs to check at least two things: (1) that the sensitive operations are correctly implemented in the DBMS and, (2) that the query interfaces are properly used by the web scripts. These checks must be performed manually.

The above ad hoc approaches have several drawbacks. First, they encourage that a significant portion of the business logic be implemented in the DBMS, which makes the interfaces less modular and less reusable. For example, we may require that only the last 4 digits of the credit card number can be displayed to the user. If this limitation is implemented in a procedure call in the DBMS, the web script can only get the last 4 digits of the number using that call. Security is guaranteed, but the web script cannot use the full number to perform further queries to the database. Instead, a dedicated procedure for this web script must be written in the database that connect these queries together. As the business logic becomes complex, the procedures in the DBMS become hard to manage.

Second, doing the dynamic checks may be inefficient. For example, the user inputs in web forms usually have constraints on them that restrict their data ranges and data types. Such constraints are often dynamically checked multiple times by the web scripts, the DBMS procedures, and the triggers in the databases. The reason is that the programming interface between the DBMS and the application software is either untyped or dynamically typed, and many constraints cannot be easily expressed as data types.

Third, the intended security policy of the system is not apparent from its implementation. There is no explicit description of what data is considered to be confidential or what the requirements are for checking the validity of untrusted inputs. This makes the software more fragile (local changes to the system can be inconsistent with the desired global security policy) and much harder to maintain over time.

1.1 Contributions

This paper proposes an language-based solution to the above problems. Instead of implementing all the access control mechanisms in the DBMS procedures or dynamic checks, we allow the programmer to specify security policies on the application programming interfaces of the DBMS. Such interfaces are strongly typed. The security policies are statically enforced in the scripting language using an information-flow type system.

Following the ideas of recent language prototypes such as Jif [7], which extends Java, and FlowCaml [14] [9] [10], which extends Caml, we design a security-typed language suitable for doing web scripting. Web scripting languages differ from general-purpose programming languages in many aspects, several of which simplify our information-flow analyses. For example, most web scripts contain little or no state, and very limited looping constructs; they involve little computation and are intended to terminate quickly. These domain-specific features allow us to deal with covert channels (such as the timing channels) and some side effects more easily than in general purpose languages.

Importantly, when we reason about the information flows in such web scripts, downgrading is very common. Many web scripts read sensitive data from databases and release them to the end user (i.e. declassification of confidential data). Conversely, web scripts also take untrusted user inputs and use them to synthesize database queries which can alter trusted data (i.e. endorsement of low integrity data). To make information-flow control effective, downgrading must be controlled in a safe manner.

Our contributions include several important extensions to prior research [5] [12]. First, we present a simple yet general architecture for building secure web systems; this paper emphasizes the scripting-language component of that architecture. Second, our language addresses the problem of downgrading in information-flow systems by providing a practical instance of a theoretical framework proposed by Li and Zdancewic [5]. Third, we extend the framework to include integrity policies and conditional downgrading policies that use a novel pattern-matching sublanguage to express information-flow constraints. Conditional downgrading policies are extremely useful for specifying requirements that involve run-time identity tests.

The rest of the paper is organized as the following. Section 2 discusses a layered architecture for information flow control in online information systems. Section 3 presents a language of security policies, defines the security levels and formalize downgrading. Section 4 presents the abstract syntax of the web scripting language, shows an program example and introduces its type system. Section 5 discusses issues with untrusted code and related work. Section 6 concludes.

2 Mandatory Access Control in Online Information Systems

2.1 System architecture

Typical web-based systems consist of a database, a database query language and various components such as scripts and web servers that work together in a multi-tiered fashion. These systems can have massive amounts of confidential and trusted information, with quite complex security policies. For example, recent USA laws mandate complicated procedures for releasing medical information about patients; such policies must be enforced in the information management systems used by health care offices. Given the ubiquity and familiarity of the web, providing a web interface to such information would be valuable, but enforcing the appropriate security policies is a challenging task.

We propose a simplified, yet general architecture for such web-based information systems, as depicted in Figure 1. The key elements are:

- The storage layer: a database management system that stores and manages confidential and trustworthy data using database schemas, tables, views and a database query language.
- The access control layer: a query interface language that controls the release and update of information in the database. A query interface provides means to access the database, but also specifies language-level information-flow policies on the interface.
- The presentation layer: a web scripting language that executes the queries, manipulate the results and generate web pages for end users. The information flow policies from the query interfaces are enforced in the web scripting language.

It is desirable to use security-typed languages to program these systems, because language-based information flow control provides strong end-to-end security guarantees [13]: information-flow policies are specified at the end-points of the program (on variables and module interfaces) and enforced in the whole program. However, in these practical systems, confidential data lives outside the programs that manipulate them. The source of such data is not program variables or function call interfaces, but rather the databases themselves. To enforce information flow policies across the whole system, it is necessary to specify the policies directly at the very end of the system, namely, at the database level, and enforce these information flow policies across the boundaries of different components.

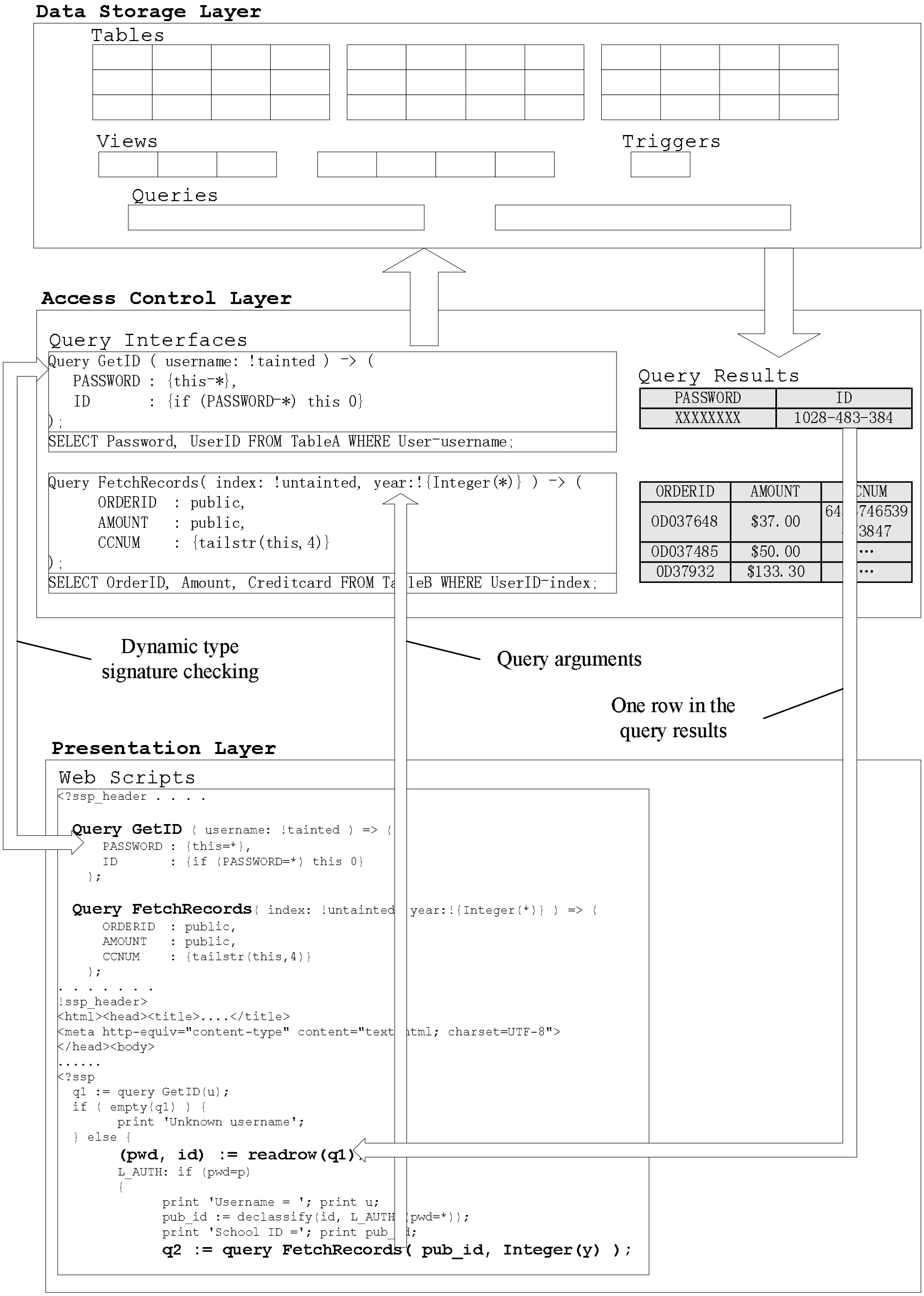
The following sections outline how the pieces of this architecture fit together; subsequent sections explain the scripting language in more detail.

2.2 Policy specification in the DBMS interface

We require that all database queries are accessed via strongly typed programming interfaces provided by the DBMS. This is shown in Figure 1 as the access control layer, which, for this example contains two queries `GetID` and `FetchRecords`. The application programmer can use only the declared interfaces to access the database but cannot execute arbitrary SQL queries. As described in the introduction, this is a standard engineering practice, as it mandates that all the security-related queries be implemented as procedure calls in the DBMS layer. However, because our scripting language types include security policy information, this layer does not have to *enforce* all the security policies in the DBMS. Instead, it can *specify* security policies as types on the output of the queries and let the application software enforce these policies using language-based information-flow control. For example, the programmer can declare the output of a query to have security level "secret", so the web scripts using this query will not leak its information to public places.

One issue is how to propagate the typing information from databases queries to the scripting language. In relational databases, a query returns a table that has a fixed number of columns. The type of each column is statically known. The script reads the query results iteratively in a row-by-row fashion, so the schema of a table can be translated to a tuple type that represents the data types for each column. As shown below, our scripting language provides built-in constructs for accessing rows of query results.

In our architecture, information-flow policies are specified on the query interface types. For confidentiality, we can annotate the output as secret so that it is not allowed to be leaked to public places. For integrity, we



```
Query GetID ( username: !tainted ) -> (
   PASSWORD : {this-*},
   ID       : {if (PASSWORD-*) this 0}
);
SELECT Password, UserID FROM TableA WHERE User-username;
```

```
Query FetchRecords( index: !untainted, year:!{Integer(*)} ) -> (
      ORDERID  : public,
      AMOUNT   : public,
      CCNUM    : {tailstr(this,4)}
);
SELECT OrderID, Amount, Creditcard FROM Ta leB WHERE UserID-index;
```

PASSWORD	ID
XXXXXXXX	1028-483-384

ORDERID	AMOUNT	CNUM
0D037648	$37.00	64 746539 3847
0D037485	$50.00	…
0D37932	$133.30	…

```
<?ssp_header . . . .

  Query GetID ( username: !tainted ) => (
      PASSWORD : {this=*},
      ID       : {if (PASSWORD=*) this 0}
    );

  Query FetchRecords( index: !untainted  year:!{Integer(*)} ) => (
      ORDERID  : public,
      AMOUNT   : public,
      CCNUM    : {tailstr(this,4)}
    );
. . . . . . .
!ssp_header>
<html><head><title>....</title>
<meta http-equiv="content-type" content="text tml; charset=UTF-8">
</head><body>
......
<?ssp
  q1 := query GetID(u);
  if ( empty(q1) ) {
        print 'Unknown username';
  } else {
        (pwd, id) := readrow(q1);
        L_AUTH: if (pwd=p)
        {
              print 'Username = '; print u;
              pub_id := declassify(id, L_AUTH (pwd=*));
              print 'School ID ='; print pub_ d;
              q2 := query FetchRecords( pub_id, Integer(y) );
```

Figure 1: Information-flow Control in Web-based Information Systems

can annotate the argument of a query as untainted so that its value is not affected by tainted inputs. Besides these built-in levels, programmers can specify *downgrading policies* as security levels that describe how secret values can become public and how untrusted values become trusted. Section 3 presents a generalized framework of these information flow policies.

2.3 Policy enforcement in the web scripting language

This remainder of this paper presents the design of a web scripting language similar to PHP, which is used for the presentation layer shown in Figure 1. The major difference with PHP is that the queries to databases are strongly typed. The arguments and results of queries are annotated with information-flow policies. The interface to each query is explicitly defined in the program, which makes auditing the software for security and maintenance purposes easier for humans and provides the information needed by our type checker to determine whether the web script satisfies the policies. At run-time, when the script connects to the DBMS, the type signature of each query interface is matched against the interface provided in the DBMS.

At a high level, a web script takes some input data provided by interaction with the user and a DBMS and produces HTML data as output. The input data coming from the client web browser can be treated as having the security labels public and tainted. Results obtained from database queries have security policy specifications defined by the query interfaces. The web script performs some computation using these inputs from different security levels and produce strings that constitute the resulting web page, which has security level public. Additionally, the query arguments sent along with the query are also outputs and they may have required security levels such as untainted. In our model, the security policies on the outputs of the script are enforced by an information-flow type system, which provides end-to-end security guarantees on the input–output relationship of the program. The following sections present our scripting language features in detail.

3 Security Levels for Confidentiality and Integrity

In many conventional information-flow type systems, the security levels considered are limited to the simple lattices public$\sqsubseteq$secret for confidentiality and untainted$\sqsubseteq$tainted for integrity. Our scripting language interprets these security levels in a more general framework, where *downgrading policies* [5] are used to express the security levels of data. A security level is simply a non-empty set of downgrading policies, where each policy describes the necessary computation that must be performed in order to downgrade the data at this security level. Such policies are expressed in a small pattern-matching language. Confidentiality policies and integrity policies are mostly symmetric in our language. The following subsections present the syntax and the semantics of these security levels.

3.1 Input variables and the pattern language

Each program takes some input data and produce some outputs. We classify the program inputs into two kinds:

- Untrusted input: inputs with security level tainted and public. Untrusted inputs come from the user inputs via web forms.
- Trusted input: inputs with security level untainted and secret, or other predefined security levels. Trusted inputs come from database query interfaces and run-time API calls. In our scripting language, such inputs are represented as *trusted variables*. For example, the root password can be modeled as a secret and untainted input to the login process. We use the metavariable s to represent such trusted input variables.

The syntax of security labels is defined in Figure 2. Each label has its confidentiality part and integrity part. Each confidentiality (or integrity) label is a non-empty set of policies. Special names such as secret, public, untainted and tainted can also be used as labels; they can be interpreted as sets of policies as shown in Figure 3. The next two subsections explain the semantics of these labels.

Label	$l ::=$	$cl \; ! \; il$
Confidentiality label	$cl ::=$	$\{cp_1, \ldots, cp_k\} \mid \mathsf{secret} \mid \mathsf{public}$
Integrity label	$il ::=$	$\{ip_1, \ldots, ip_k\} \mid \mathsf{tainted} \mid \mathsf{untainted}$
Confidentiality policy	$cp ::=$	$c \mid s \mid \star \mid \oplus\, cp_1, \ldots, cp_k \mid \mathsf{if}\ cp\ cp\ cp \mid \mathsf{this}$
Integrity policy	$ip ::=$	$c \mid s \mid \star \mid \oplus\, ip_1, \ldots, ip_k \mid \mathsf{if}\ ip\ ip\ ip$
Trusted variable	$s ::=$	s_i
Constant	$c ::=$	c_i (constant integers and strings)
Operator	$\oplus ::=$	$+ \mid - \mid \mathsf{hash} \mid$ built-in operators...
Downgrading actions	$a ::=$	$c \mid s \mid \star \mid \oplus\, a_1, \ldots, a_k \mid \mathsf{if}\ a\ a\ a \mid \mathsf{this}$

Figure 2: The Pattern Language for Security Labels

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{secret} & \equiv & \{c_0\} & \mathsf{public} & \equiv & \{\mathsf{this}\} \\ \mathsf{tainted} & \equiv & \{\star\} & \mathsf{untainted} & \equiv & \{ip \mid \star \notin ip\} \end{array}$$

Figure 3: Interpretation of Conventional Security Labels

3.2 Confidentiality labels

Syntactically, a **confidentiality policy** is an expression embedded with this and $\star$. The meaning of such a policy is some computation required to declassify (or downgrade) the secret. When this is replaced by the annotated secret value and every $\star$ is replaced by a value at security level public, the result of the expression is at public. For example, if the password p has a policy "this $= \star$" and x is a public value, then the expression p=x is a public value. This policy allows the password be compared to untrusted values, but does not allow the password be leaked by other means.

Policy equivalence: We write $cp_1 \equiv cp_2$ to denote that the two policies have the same semantic meaning. For example, $x + y \equiv y + x$. To keep things simple in this paper, we simply use syntactical equivalence.

A **confidentiality label** is a non-empty set of confidentiality policies. Such a label describes a security level for confidentiality. If data x has a confidentiality label $cl = \{cp_1, \ldots, cp_k\}$, it means x can only be downgraded to public by using one of the policies cp_i in that label. In other words, a label specifies possible ways (policies) to downgrade the data it annotates. For example, the label $\{\mathsf{this} = \star, \mathsf{hash}(\mathsf{this})\}$ on a password value also allows the password be leaked by computing its hash. We define the **interpretation** of a label cl to be an infinite set of policies:

$$\mathbb{S}(cl) \stackrel{\triangle}{=} \{cp_1 \mid cp_1 \equiv [cp_2/\mathsf{this}]cp_3,\ cp_2 \in cl, s_i \notin cp_3\}$$

Intuitively, it means that if cp_2 is a valid downgrading policy, i.e. cp_2 represents a public expression, we can put cp_2 in another context cp_3 to yield another downgrading policy cp_1, as long as the context cp_3 does not contain trusted secret variables. The interpretation of a label is the set of all policies that can be derived from the policies in that label. For example, if "hash(this)" is a valid downgrading policy, then it implies the policy "hash(this)%16" is also valid.

In this framework, both secret and public can be expressed using confidentiality policies: secret can be represented by a constant policy $\{c_0\}$, which trivially hides the secret after substituting this with the secret. The interpretation of secret is all the pattern expressions that does not contain this and secret variables. The security level public can be represented by the expression $\{\mathsf{this}\}$, which says the annotated value can be straightforwardly treated as public data. The interpretation of public includes all the pattern expressions that does not contain trusted secret variables. The **ordering** among labels is then defined using set inclusion:

$$cl_1 \sqsubseteq cl_2 \iff \mathbb{S}(cl_2) \subseteq \mathbb{S}(cl_1)$$

As an example, it is easy to verify that $\mathsf{public} \sqsubseteq \{\mathsf{this} = \star, \mathsf{hash}(\mathsf{this})\} \sqsubseteq \{\mathsf{this} = \star\} \sqsubseteq \mathsf{secret}$. Intuitively, higher security levels contains fewer downgrading policies. The **join** of labels is interpreted as the join of the label interpretations, which can be conservatively approximated by taking the intersection of two labels.

These security labels allow us to formally define **downgrading** in programs. We use the metavariable a to represent **actions**, which specifies a fragment of computation on a value. In an action expression, this denotes the value of interest and all the $\star$ represent values of public level. A label cl can be downgraded to another label using an action a:

$$\Downarrow (cl, a) = \{cp_1 \mid [a/\mathsf{this}]cp_1 \equiv cp_2,\ cp_2 \in cl\}$$

Here, $\Downarrow (cl, a)$ is the label of data obtained by taking data with label cl and performing action a on it. For example, suppose we define the following labels and actions:

$$\begin{array}{ll} cl_1 \triangleq \{(\mathsf{hash}(\mathsf{this})\%4){=}\star\} & a_1 \triangleq \mathsf{hash}(\mathsf{this}) \\ cl_2 \triangleq \{(\mathsf{this}\%4){=}\star\} & a_2 \triangleq \mathsf{this}\%4 \\ cl_3 \triangleq \{\mathsf{this}{=}\star\} & a_3 \triangleq \{\mathsf{this}{=}\star\} \end{array}$$

We have $\Downarrow (cl_1, a_1) = cl_2$, $\Downarrow (cl_2, a_2) = cl_3$, $\Downarrow (cl_3, a_3) = \mathsf{public}$. If a variable x has security level cl_1 and y is public, then the expression $\mathsf{hash}(x)$ has level cl_2, $\mathsf{hash}(x)\%4$ has level cl_3, $(\mathsf{hash}(x)\%4) = y$ has level public. Intuitively, an action is a pattern that matches the computation in the program: this matches a secret value to be downgraded and $\star$ matches any expressions at level public. Each confidentiality label corresponds to a state machine that models downgrading, where states are labels and transitions are downgrading actions.

3.3 Integrity labels

Integrity labels are largely the dual of confidentiality labels with several subtle differences. Confidentiality policies specify what can be done with the data in the future, integrity policy specify what has been done to the data in the past. An **integrity policy** is an expression embedded with $\star$, meaning an expression that has computed the annotated value as a result, where each $\star$ represents an untrusted, tainted input to the expression. For example, if the variable x is tainted, then the expression $x\%4$ has an integrity policy "$\star\%4$", which states an integrity constraint on the result. An **integrity label** is a non-empty set of integrity policies, where each policy describes an expression that could have computed the value as a result. If the value x has an integrity label $il = \{ip_1, \ldots, ip_k\}$, then it must have been computed using one of the expression $ip_j \in il$. Most interesting integrity labels have only one policy in them, because adding policies to a label only *weakens* the integrity guarantee. Similar to confidentiality labels, the **interpretation** of integrity labels is defined as:

$$\mathbb{S}(il) \triangleq \{ip_1 \mid ip_1 \equiv [ip_2/\star_j]ip_3,\ ip_3 \in il\}$$

The **ordering** among integrity labels is defined as:

$$il_1 \sqsubseteq il_2 \iff \mathbb{S}(il_1) \subseteq \mathbb{S}(il_2)$$

In this framework, tainted can simply be represented as $\{\star\}$, as this is dual case of public in confidentiality. The security label untainted corresponds to an infinite label $\{ip \mid \star \notin ip\}$, which includes all expressions that do not use tainted inputs. Although we have $\mathsf{untainted} \sqsubseteq \mathsf{tainted}$, many interesting policies do not sit between these two security levels. For example, $\{\mathsf{min}(\star, 10)\}$ has no direct ordering with untainted. This explains why traditional definitions of noninterference gives a weak form of integrity guarantee: there are many interesting integrity policies besides untainted. In fact, untainted is a very coarse security level and it can be further strengthened. For example, the integrity label $il_1 \triangleq \{s_2 + s_3\}$ satisfies $il_1 \sqsubseteq \mathsf{untainted}$. It says a very strong integrity guarantee: values at this integrity level must be equal to the sum of two trusted program inputs. An even stronger integrity label $\{0\}$ works like a singleton type where the only inhabitant is zero. **Downgrading** for integrity labels is formalized as:

$$\Downarrow (il, a) \triangleq \{ip_1 \mid [ip_2/\mathsf{this}]a \equiv ip_1,\ ip_2 \in il\}$$

For example, suppose

$$\begin{array}{ll} a_1 \triangleq \mathsf{ToInt}(\mathsf{this}) & il_1 \triangleq \{\mathsf{ToInt}(\star)\} \\ a_2 \triangleq \mathsf{min}(\mathsf{this}, 10) & il_2 \triangleq \{\mathsf{min}(\mathsf{ToInt}(\star), 10)\} \\ a_3 \triangleq \mathsf{max}(\mathsf{this}, 5) & il_3 \triangleq \{\mathsf{max}(\mathsf{min}(\mathsf{ToInt}(\star), 10), 5)\} \end{array}$$

We have $\Downarrow(\mathsf{tainted}, a_1) = il_1$, $\Downarrow(il_1, a_2) = il_2$, $\Downarrow(il_2, a_3) = il_3$. If a variable x has security level il_2, then the expression $\mathsf{max}(x, 5)$ has level il_3.

4 The Web Scripting Language

4.1 Language syntax and semantics

$$\begin{array}{lrl}
\text{Variables} & x ::= & x_i \\
\text{Expressions} & e ::= & c \mid s \mid x \mid \oplus\, e_1, \ldots, e_k \mid \mathsf{declassify}(e, L : a) \mid \mathsf{endorse}(e, L : a) \\
\text{Commands} & p ::= & \epsilon \mid p; p \mid x := e \mid [L :]\mathsf{if}\ e\ p\ p \mid \mathsf{while}\ e\ p \mid \mathsf{print}\ e \\
 & & \mid x := \mathsf{query}\ name\ (e_1, \ldots, e_k) \mid (s_1, \ldots, s_k) := \mathsf{readrow}\ x \\
 & & \\
\text{Types} & \tau ::= & \mathsf{string}\ l \mid \mathsf{query}\ name \\
 & & \\
\text{Programs} & Prog ::= & Inputs\ Queries\ Vars\ Boby \\
\text{Input} & Input ::= & field \Rightarrow s \\
\text{Query} & Query ::= & name\ (x_1 : il_1, \ldots, x_j : il_j) \Rightarrow (s_1 : cl_1, \ldots, s_k : cl_k) \\
\text{Variables} & Var ::= & x : cl\ !\ il \\
\text{Body} & Body ::= & c\ Body \mid p\ Body \mid \epsilon
\end{array}$$

Figure 4: Abstract Syntax

The web scripting language provides a programming model similar to PHP. The abstract syntax is shown in Figure 4. Program fragments are inserted to the web page using special tags like `<?ssp ...  !ssp>`. At the top level, a web script consists of a header and several program fragments. A header includes the mapping from the form inputs to variable names and the definition of query interfaces. A query interface includes the name of the query, the query arguments, the result variables and the security levels of these variables. Each program fragment is a command. Commands can be sequential composition of commands, assignments, branches, loops, print statements, and query operations. For simplicity, function calls are not presented in this paper, although they are not fundamentally difficult to include. One other difference from PHP and other information-flow languages is that a branch statement can have a tag L on it; such tags can be used in declassify and endorse expressions, which is explained later. When the script is executed by the web server, each program fragment is substituted by its output using the print statement. Except query handles, all values are simply strings in this language.

The type system of the web scripting language is presented in Figures 5, 6 and 7. The type system statically controls the information flow in the programs. Confidentiality labels and integrity labels are tracked separately. Like Jif and FlowCaml, the type system generally disallows information flow from high security levels to low security levels, where the ordering of security labels is defined as in Section 3. For confidentiality, implicit information flows are also tracked by adding a program counter label *pc* to the typing context. The *pc* label is only permitted to be either public or secret, because the downgrading policies written by the programmer apply only to expressions, not to control flow.

A typing judgment for an expression has the form $\Gamma, \Phi \vdash e : \tau$, where Γ is the typing context for variables and Φ is the context of conditional expressions, explained below. The typing judgments for commands are of the form $\Gamma, \Phi, pc \vdash p$ meaning that the command p is well-typed under the contexts Γ, Φ and the program counter *pc*. For example, the C-Assign rule only allows information flow from low security levels to high security levels. It also take *pc* label into account to track implicit flows.

The program writes its output using the `print` statement. Output data is publicly visible to the user, so the type system must ensure that the confidentiality label of the output is public in the C-Print rule.

$$\frac{\Gamma, \Phi, pc \vdash p_1 \qquad \Gamma, \Phi, pc \vdash p_2}{\Gamma, \Phi, pc \vdash p_1; p_2} \qquad \textsc{C-Composition}$$

$$\frac{\begin{array}{c}\Gamma(x) = \textsf{string } cl_1 \ ! \ il_1 \\ \Gamma, \Phi \vdash e : \textsf{string } cl_2 \ ! \ il_2 \qquad cl_2 \sqsubseteq cl_1 \sqcup pc \qquad il_2 \sqsubseteq il_1\end{array}}{\Gamma, \Phi, pc \vdash x := e} \qquad \textsc{C-Assign}$$

$$\frac{\Gamma, \Phi \vdash e : \textsf{string } cl \ ! \ il \qquad cl \equiv \textsf{public} \qquad pc \equiv \textsf{public}}{\Gamma, \Phi, pc \vdash \textsf{print } e} \qquad \textsc{C-Print}$$

$$\frac{\begin{array}{c}\Gamma, \Phi \vdash e : \textsf{string } cl \ ! \ il \\ cl \equiv \textsf{public} \qquad \Gamma, \Phi, \textsf{public} \vdash p \qquad pc \equiv \textsf{public}\end{array}}{\Gamma, \Phi, pc \vdash \textsf{while } e \ p} \qquad \textsc{C-While}$$

$$\frac{\begin{array}{c}\Gamma, \Phi \vdash e : \textsf{string } cl \ ! \ il \qquad cl \sqcup pc \sqsubseteq \textsf{public} \\ \Gamma, \Phi \cup (L^+ : e), \textsf{public} \vdash p_1 \qquad \Gamma, \Phi \cup (L^- : e), \textsf{public} \vdash p_2\end{array}}{\Gamma, \Phi, pc \vdash L : \textsf{if } e \ p_1 \ p_2} \qquad \textsc{C-If-Pub}$$

$$\frac{\begin{array}{c}\Gamma, \Phi \vdash e : \textsf{string } cl \ ! \ il \\ \Gamma, \Phi, \textsf{secret} \vdash p_1 \qquad \Gamma, \Phi, \textsf{secret} \vdash p_2\end{array}}{\Gamma, \Phi, pc \vdash L : \textsf{if } e \ p_1 \ p_2} \qquad \textsc{C-If-Sec}$$

$$\frac{\begin{array}{c}Q(name) = (x_1 : il_1, \ldots, x_k : il_k) \Rightarrow (\ldots) \\ \Gamma, \Phi \vdash e_i : \textsf{string } cl_i \ ! \ il'_i \qquad cl_i \equiv \textsf{public} \\ il'_i \sqsubseteq il_i \qquad pc \equiv \textsf{public} \qquad \Gamma \cup (x : \textsf{query } name), \Phi, pc \vdash p\end{array}}{\Gamma, \Phi, pc \vdash x := \textsf{query } name \ (e_1, \ldots, e_k); p} \qquad \textsc{C-Query}$$

$$\frac{\begin{array}{c}\Gamma(x) = \textsf{query } name \\ Q(name) = (\ldots) \Rightarrow (S_1 : cl_1, \ldots, S_k : cl_k) \\ cl'_i \stackrel{\triangle}{=} pc \sqcup [s_1/S_1] \ldots [s_k/S_k] cl_i \\ pc \equiv \textsf{public} \qquad \Gamma \cup (s_i : \textsf{string } cl'_i \ ! \ \{s_i\}), \Phi, pc \vdash p\end{array}}{\Gamma, \Phi, pc \vdash (s_1, \ldots, s_k) := \textsf{readrow } x; p} \qquad \textsc{C-ReadRow}$$

Figure 5: Command Typing Rules: $\boxed{\Gamma, \Phi, pc \vdash p}$

$$\frac{}{\Gamma, \Phi \vdash c : \mathsf{string}\ \mathsf{public}\ !\ \{c\}} \qquad \textsc{E-Const}$$

$$\frac{\Gamma(s) = \mathsf{string}\ cl\ !\ il}{\Gamma, \Phi \vdash s : \mathsf{string}\ cl\ !\ il} \qquad \textsc{E-TrustVar}$$

$$\frac{\Gamma(x) = \mathsf{string}\ cl\ !\ il}{\Gamma, \Phi \vdash x : \mathsf{string}\ cl\ !\ il} \qquad \textsc{E-Var}$$

$$\frac{\Gamma, \Phi \vdash x : \mathsf{query}\ name}{\Gamma, \Phi \vdash \mathsf{empty}(x) : \mathsf{string}\ \mathsf{public}\ !\ \mathsf{tainted}} \qquad \textsc{E-EmptyTest}$$

$$\frac{\begin{array}{cc}\Gamma, \Phi \vdash e_1 : \mathsf{string}\ cl_1\ !\ il_1 & \Gamma, \Phi \vdash e_2 : \mathsf{string}\ cl_2\ !\ il_2 \\ CL(\oplus, cl_1, cl_2, il_1, il_2) = cl_3 & IL(\oplus, cl_1, cl_2, il_1, il_2) = il_3\end{array}}{\Gamma, \Phi \vdash e_1 \oplus e_2 : \mathsf{string}\ cl_3\ !\ il_3} \qquad \textsc{E-Op}$$

$$\frac{\begin{array}{c}\Gamma, \Phi \vdash e_1 : \mathsf{string}\ cl_1\ !\ il \\ \Phi(L^+) = e_2 \qquad \mathsf{match}(\Gamma, e_2, a) \qquad cl_2 \overset{\triangle}{=}\Downarrow (cl, \mathsf{if}\ a\ \mathsf{this}\ 0)\end{array}}{\Gamma, \Phi \vdash \mathsf{declassify}(e_1, L : a) : \mathsf{string}\ cl_2\ !\ il} \qquad \textsc{E-Declassify+}$$

$$\frac{\begin{array}{c}\Gamma, \Phi \vdash e_1 : \mathsf{string}\ cl_1\ !\ il \\ \Phi(L^-) = e_2 \qquad \mathsf{match}(\Gamma, e_2, a) \qquad cl_2 \overset{\triangle}{=}\Downarrow (cl, \mathsf{if}\ a\ 0\ \mathsf{this})\end{array}}{\Gamma, \Phi \vdash \mathsf{declassify}(e_1, L : a) : \mathsf{string}\ cl_2\ !\ il} \qquad \textsc{E-Declassify-}$$

Figure 6: Expression Typing Rules: $\boxed{\Gamma, \Phi \vdash e : \tau}$

Furthermore, the *pc* label in the typing context of the `print` statement must also be public to prevent implicit information flow such as "`if secret then print 1 else print 0`". Secret data must be downgraded to public data before they can be printed to web pages.

Figure 8 shows an actual web script. It has two query interfaces to the database. The script takes the user input, performs queries to the database, reads the results from the query and generates a web page for the end user. The following subsections walk through this example step-by-step to present the language features.

4.2 Query interfaces and type declarations

Query interfaces and variables are declared in the header section of a script. In Figure 8, the script header from line 1 to line 17 specifies all the input/output data types. Line 2 specifies the inputs fields submitted from the web forms in a HTTP request and maps them to variables `u`, `p` and `y` in the scripting language. All of these variables are considered to be public, untrusted data; they have security level "public ! tainted" in the type system.

Two query interfaces are defined in lines 4-13. At run-time, they must match their specifications in the DBMS interfaces. The first query, `GetID`, looks up a user name in the database and returns the user's password and identity number; both are secrets and cannot be directly released to the public. The second query `FetchRecords` uses the user's identity number to look up the user's transaction history in the database.

Query arguments: Integrity labels are specified for query arguments in the interface, their confidentiality labels are public by default as required by the C-Query typechecking rule. For example, the query argument `index` on line 9 has an integrity label untainted, which makes it impossible to pass an untrusted tainted value to `index`. The argument `year` on line 9 requires a mandatory conversion from an arbitrary

$$\frac{}{CL(\oplus, cl_1, cl_2, il_1, il_2) = \mathsf{secret}} \qquad \text{CL-SECRET}$$

$$\frac{cl_1 \sqsubseteq \mathsf{public} \qquad cl_2 \sqsubseteq \mathsf{public}}{CL(\oplus, cl_1, cl_2, il_1, il_2) = \mathsf{public}} \qquad \text{CL-PUBLIC}$$

$$\frac{\begin{array}{c} il_2 \sqsubseteq \{ip\} \\ a_1 \overset{\triangle}{=} \mathsf{this} \oplus ip \qquad \mathsf{match}(a_1, a_2) \qquad cl_3 \overset{\triangle}{=} \Downarrow (cl_1, a_2) \end{array}}{CL(\oplus, cl_1, \mathsf{public}, il_1, il_2) = cl_3} \qquad \text{CL-DOWNGRADE(L)}$$

$$\frac{}{IL(\oplus, il_1, il_2) = \mathsf{tainted}} \qquad \text{IL-TAINTED}$$

$$\frac{il_1 \sqsubseteq \mathsf{untainted} \qquad il_2 \sqsubseteq \mathsf{untainted}}{IL(\oplus, il_1, il_2) = \mathsf{untainted}} \qquad \text{IL-UNTAINTED}$$

$$\frac{}{IL(\oplus, \{ip_1\}, \{ip_2\}) = \{ip_1 \oplus ip_2\}} \qquad \text{IL-COMPOSE}$$

Figure 7: Downgrading Rules

string to a string that contain only an integer, which forbids certain SQL string attacks.

Query results: Query results are modeled as trusted variables in the type system. All the trusted variables are treated as read only in the type system: they cannot be updated using direct assignments. Confidentiality labels are specified for query results, and their integrity labels are implicitly defined by the C-READROW rule: a variable `s` has an integrity label {`s`}. On line 5, the confidentiality label for `PASSWORD` states that the only possible way to leak information about the password is to compare it with a value at level public.

A row of the query result is read together using the `readrow` command. The confidentiality policy of a query result variable can mention names of other variables in the same query. For example, the variable `ID` on line 6 has a policy that mentions the variable `PASSWORD`, saying that the the `ID` string can only be disclosed if a publicly known string matches `PASSWORD` on the same row of the query result. This policy specifies a run-time test of identity information.

4.3 Downgrading

Downgrading is the key feature of this type system. The downgrading policies specified by the query interfaces control how confidential data is released and how trustworthy information is updated. There are two downgrading mechanisms in the type system.

Implicit downgrading: Downgrading happens implicitly in each step of computation that uses the built-in operators. Without loss of generality, we present only the typing rules for binary operators in this paper. In traditional security type systems, if x has security level l_1 and y has level l_2, the result of $x \oplus y$ has security level $l_1 \sqcup l_2$, which is an upperbound of l_1 and l_2. The E-OP rule in Figure 6 is backward-compatible with those type systems. However, E-OP examines the labels of the operands more carefully using rules in Figure 7. The CL-SECRET, CL-PUBLIC, IL-TAINTED and IL-UNTAINTED rules are standard rules—they give the label of the result by approximating the join of the arguments. The IL-COMPOSE rule, however, attempts to compute the integrity label for the result when both operands have good integrity guarantees. The CL-DOWNGRADE(L) rule declassifies the left operand using an action that corresponds to the use of the $\oplus$ operator (there is a symmetric version that operates on the right side of $\oplus$). The integrity label of the other operand is also used to describe this action.

It is possible that more than one action a_2 can be chosen in the CL-DOWNGRADE rule. For example, the

```
<?ssp_header
  FormInputs  ( "UserName" => u, "Password" => p,  "QueryYear" => y );

  Query GetID ( username: !tainted ) => (
      PASSWORD : {this=*},
      ID       : {if (PASSWORD=*) this 0}
    );

  Query FetchRecords( index: !untainted, year:!{Integer(*)} ) => (
      ORDERID  : public,
      AMOUNT   : public,
      CCNUM    : {tailstr(this,4)}
    );

  Variables ( pub_id: public!untainted );

!ssp_header>
<html><head><title>....</title>
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
</head><body>
......
<?ssp
  q1 := query GetID(u);
  if ( empty(q1) ) {
        print ’Unknown username’;
  } else {
        (pwd, id) := readrow(q1);
        L_AUTH: if (pwd=p)
        {
              print ’Username = ’; print u;
              pub_id := declassify(id, L_AUTH:(pwd=*));
              print ’School ID =’; print pub_id;
              q2 := query FetchRecords( pub_id, Integer(y) );
              while (!empty(q2)) {
                    (orderid, amoumt, ccnum) := readrow(q2);
                    print ’Order ID = ’; print orderid;
                    print ’Amount = ’; print amount;
                    print ’Credit Card = XXXX-XXXX-XXXX-’;
                    print tailstr(ccnum, 4);
              }
        } else {
              print ’Wrong password’;
        }
  }
!ssp>
.....
```

Figure 8: A Web Script Example

action $\mathsf{this} = c_0$ can match the policy $\mathsf{this} = \star$. The predicate match determines whether the action matches a downgrading policy pattern; we omit the straightforward definition of when such patterns unify. We make this downgrading implicit because most useful downgrading policies are simple and it is easy to search (in the implementation of match) for an usable downgrading action. To avoid searching, the language could be extended with an optional construct that specifies the downgrading action as annotations on the operator so that the type checker knows which action to use.

Conditional downgrading: The conditional expressions in the policy language allow us to specify downgrading patterns with branches. This is achieved in the type system by tracking the "active conditions" on the current execution path. The if statements can have an optional tag L in its syntax. In the C-If-Pub rule, the context Φ is used to keep track of all the conditional expressions tags on the current execution path. These tags are also annotated with either $+$, indicating the "true" branch, or $-$, indicating the "false" branch. Programs can use declassify and endorse statements to downgrade the label of a value by specifying the tag of the conditional expression that corresponds to the if statement, and an action that *matches* the conditional expression. Here again, a $\mathsf{match}(\Gamma, e, a)$ predicate is needed to determine whether the expression e can instantiate action a in context Γ. Intuitively, the tag L mentioned in the downgrading operation provides the justification for control flow that validates the use of the downgrading action. This is a novel extension of the type system by Li and Zdancewic [5] without breaking the *relaxed noninterference* result: the relaxed noninterference of conditional patterns can be be justified by an equivalence rule "$\mathsf{if}\ e_1\ E[e_2]\ e_3 \equiv \mathsf{if}\ e_1\ E[\mathsf{if}\ e_1\ e_2\ c]\ e_3$" where E is an evaluation context, with some side conditions on the typing contexts and variable bindings.

4.4 Information flow control in the example

In Figure 8, lines 22-44 show a program fragment in the web script. It uses the query `GetID` to authenticate the user and uses the query `FetchRecords` to look up the user's history of transactions.

Reading query results: Line 23 submits a query to the database and returns a handle `q1` to the instance of this query. Line 27 reads a row from the query results. When the variables `(pwd,id)` are added to the context, their types are added to the typing context according to the database interfaces. The variable `pwd` has security level {`this=*`}`!`{`pwd`} and `id` has security level {`if (pwd=*) this 0`}`!`{`id`}. The variable names in the query interfaces are substituted by the actual instances in the C-ReadRow rule.

Implicit downgrading: Line 28 performs an implicit downgrading in the conditional expression. The expression `pwd=p` corresponds to an action "$\mathsf{this} = \star$" for the variable `pwd`, and we have $\Downarrow(\{\mathsf{this} = \star\}, \{\mathsf{this} = \star\}) = \mathsf{public}$. Therefore, the expression `pwd=p` has security level public ! tainted by using the E-Op rule, the CL-DownGrade(L) rule and the IL-Tainted rule. The information leak on this expression is permitted by the policy and the *pc* label inside the branch will be public. In contrast, in languages like Jif, downgrading must be performed by using its `declassify` expression. Implicit downgrading also happens on line 33 for the variable `y` and on line 39 for the variable `ccnum`. The type system provides substantial guarantees about downgrading—all the downgrading must follow permissible downgrading paths specified by the policies.

Conditional downgrading: When the `if` statement is typechecked on line 28, the tag `L_AUTH` and the conditional expression `(pwd=p)` are stored in the context Φ. This information is used to justify further downgrading inside the body of the branches. Line 31 shows such an example. The declassify operation is a no-op at run-time. It merely serves as a hint to the typechecker that a conditional test is in the current execution path and the expression `id` can be downgraded using an action corresponding to the conditional test on line 28. First, the E-Declassify+ rule verifies that the action `(pwd=*)` matches the conditional expression `(pwd=p)` that correspond to the tag `L_AUTH` in the typing context. Then, the security level of `id` is downgraded using the action `(if (pwd=*) this 0)` where `0` is an arbitrary constant. Thus, the resulting confidentiality label for `pub_id` is public. If the programmer does not perform the required identity test (specified on line 6), the type system will not permit the program to output the `ID`. This conditional downgrading policy effectively enforces a run-time identity test [15].

Writing query arguments: The `declassify` statement only affects the confidentiality label. The integrity label for `pub_id` is still {`id`}. Line 33 calls another query `FetchRecords` using the value `pub_id`. According to the C-Query rule, the interface of `FetchRecords` demands that the first argument has an untainted integrity level. This is satisfied because $\{\mathtt{id}\} \sqsubseteq \mathsf{untainted}$ in our framework. If the `FetchRecords` query used a tainted value—perhaps obtained from user input—the type system will detect such the error.

5 Discussion

5.1 Untrusted code, timing channels and side effects

Our web scripting language is primarily intended as a tool to help web-systems builders create more secure systems. As such, the main focus of this paper has been on *trusted code* which is written without malicious intent. Dealing with *untrusted code* is a much more difficult problem. However, the downgrading and trust model described in this paper differs from previous work in a couple significant ways.

In Jif, downgrading is controlled using the decentralized label model, where each principal can only downgrade its own policies. The DLM uses the notion of *authority* as justification for privileged operations; but *authority* is not connected to the program semantics. As a result, untrusted code (code without the authority of a principal P) cannot downgrade data owned by P. Our language provides a complementary ability to specify downgrading policies based on *required computation* rather than using *code privileges.* This makes it possible to allow untrusted code to perform downgrading in a safe manner as long as the downgrading policies are correctly specified (and the untrusted code passes the typechecker). Of course, even for untrusted code, the downgrading policies must be specified conservatively with possible attacks in mind. For example, the `FetchRecords` query is not safe to use in an untrusted setting because the attacker can enumerate possible identity numbers and steal information from the database. This problem can be solved by posing stronger policies on the query interface, for example, using run-time identity tests like the policy on line 6.

For confidentiality, untrusted code can also leak information through covert channels such as timing channels and side effects. This problem is solved by requiring the *pc* label be `public` at all places where observable side effects are possible to happen: loops, reading rows from queries, etc. In the type system, the C-Print, C-While, C-Query and C-ReadRow all require the *pc* label to be `public`. This solution is impractical for languages like Jif and FlowCaml, because the explicit declassification needed make it too clumsy to allow useful programs be written. However, our language makes it more practical, because many secret data can be implicitly downgraded to public data before they affect the *pc* label. The *pc* label is indeed `public` everywhere in the example in Figure 8. It is also worth pointing out that the control flow in a web script is often simpler than other programs. Web scripts naturally use the continuation-passing programming style and many scripts execute for a very short time. By limiting the confidentiality label of the loop condition in the C-While rule, timing channel leaks are largely eliminated.

For trusted code, the requirement on the *pc* label can be relaxed. Instead of rejecting a program, the typechecker can raise appropriate warnings in the C-Print, C-While, C-Query and C-ReadRow rules, where confidential information can be leaked through covert channels.

The web scripting language presented in this paper is intended to be a practical instantiation of a more theoretical language developed by Li and Zdancewic. However, despite confidence in the type system presented here (as justified by that previous work), we have not proved any formal security guarantees about it. The security goal is harder to formalize than the *relaxed noninterference* result proved previously, because this language includes side effects and state. A promising future direction is to formalize the security guarantee for this language, perhaps by using a functional variant of this language with monadic effects.

5.2 Related Work

Language-based information-flow control has been studied for some time [13]. Recent language prototypes such as Jif [7], which extends Java, and FlowCaml [14], which extends Caml, provide nonstandard type systems that enforce information-flow policies. The security guarantee of such type systems is usually formalized as *noninterference* [4] [6], an end-to-end extensional guarantee that requires that no information propagates from high security levels to low security levels. However, there have been very few practical applications that demonstrate the use of these security-typed languages. In this paper, we have proposed to apply these techniques to web scripting languages, for which security concerns are increasingly important.

The current state-of-the art in web scripting protects data confidentiality and (perhaps more importantly) integrity in ad hoc ways. Here we aim to do better, yet still provide a practical enforcement mechanism.

One important challenge in making such an approach practical is the problem of *downgrading* [8] [16] [3] [11] [2] [5]. Noninterference alone is too strong for practical use. For confidentiality, it is usually necessary that secret data can be leaked to public places, but only in controlled ways. One approach, the decentralized label model (DLM) [8] can control downgrading by using privileges associated with the code. The DLM allows us to specify policies about *who* can downgrade the data, but does not specify *how* the data should be downgraded and *what* what is downgraded. As a result, the end-to-end noninterference guarantee no longer holds for code with downgrading.

Another problem is information *integrity policies* [1]. Although confidentiality and integrity are usually considered as duals in information-flow systems, the resulting integrity guarantee is weak. Noninterference guarantees only that tainted data does not affect the values of the trusted, untainted data, but it does not say anything about how the trusted data are manipulated in the system. There is absolutely no integrity guarantee for data coming from untrusted code in the Jif language, because a malicious program can manipulate the trusted data in arbitrary ways without using tainted data. As a result, the two-dimensional DLM degenerates to a one-dimensional *trust model* or *writers model* for integrity policies.

Recent advances in the research on *downgrading* extend the notion of noninterference by specifying *downgrading policies* as security levels and studies *how* the data are downgraded. *Relaxed noninterference* [5] and *delimited release* [12] provide end-to-end security guarantees on downgrading. In the theoretical framework proposed by Li and Zdancewic [5] , an information-flow type system is used to control downgrading in a fine-grained manner according to the *downgrading policies* specified by the programmer. A secure program can be proved to be equivalent to a special form where all the downgrading are explicit and external to the body of the program. The security guarantee can then be interpreted in the model of *delimited release* [12] proposed by Sabelfeld and Myers, which states a weakened and backward-compatible version of noninterference.

This paper integrates these theoretical frameworks together in a practical, domain-specific programming language. Li and Zdancewic express the downgrading policies as lambda calculus terms; manipulating these policies requires higher-order unification and extensive proof searching. In this paper, we simplify the policy language by using patterns to represent policies and use straightforward pattern matching in type-checking to avoid extensive searching. Instead of using an effect type system to enforce delimited release [12], we simply require that all the confidential input variables are read-only variables. Furthermore, the conditional downgrading policies can be used to enforce run-time identity tests and achieve similar goals with *run-time principals* [15].

6 Conclusion

This paper presents an architecture for obtaining strong, end-to-end security in web-based online information systems and motivated the use of language-based information-flow control in a web scripting language. In this approach, information-flow policies are specified in the database query interfaces and enforced in the web scripting language by a static type checker.

Based on prior research, this paper presents a framework of downgrading policies using a simple and tractable pattern language that connects implicit downgrading to computations in the script. Integrity policies and confidentiality policies are treated symmetrically, leading to a clean and intuitive way for programmers to describe their policies. Moreover, this paper presents a novel downgrading mechanism by that works by tracking the conditional expressions in the typing context and using them to enforce policies on run-time conditions such as identity tests.

References

[1] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[2] Stephen Chong and Andrew Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004.

[3] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. 31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 186–197, Venice, Italy, January 2004.

[4] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[5] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. 32th ACM Symp. on Principles of Programming Languages (POPL)*, 2005.

[6] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

[7] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[8] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[9] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, September 2000.

[10] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, January 2002.

[11] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. of the 12th IEEE Computer Security Foundations Workshop*, 1999.

[12] Andrei Sabelfeld and Andrew Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security (ISSS'03)*, 2004.

[13] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[14] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, March 2003.

[15] Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *Proc. IEEE Symposium on Security and Privacy*, 2004.

[16] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 268–276. ACM Press, January 2000.