Cross-tier, Label-based Security Enforcement for Web Applications

Brian J. Corcoran Nikhil Swamy Michael Hicks

University of Maryland, College Park
{bjc, nswamy, mwh}@cs.umd.edu

ABSTRACT

This paper presents SELinks, an extension of the Links web programming language, that allows a database and web server to collaboratively enforce a security policy with high assurance. Our approach has a number of benefits. First, the relationship between data and its security label is made explicit by the SELinks type system, which allows the compiler to ensure that a policy is always correctly enforced. Next, application-specific logic is communicated seamlessly to the database by compiling SELinks code and values to user-defined functions and custom datatypes, respectively, to be stored in the database. As a result, application-specific security policies can be enforced at the database while processing queries, improving both the overall efficiency of the application, as well as ensuring that sensitive data never leaves the database needlessly. Our experience with two sizeable web applications indicates that cross-tier policy enforcement in SELinks is flexible, relatively easy to use and improves efficiency, in terms of increased throughput, by as much as an order of magnitude.

1. INTRODUCTION

Multi-tier web applications increasingly require fine-grained security policies. As an example, consider a system for on-line access to medical records, where each part of a record may be subject to different policy concerns. Most parts of a record may be read by the patient and her primary doctor, but an insurance provider may only read information relevant to billing, and a lab may only write test results but not otherwise read the record. Access policies such as these are typically expressed as potentially-complex metadata associated with a relevant part of a record stored within the database. We think of this policy metadata as a *security label*—for instance, a label could be an access control list containing names of authorized users. On-line stores, web portals, e-voting systems, and collaborative information sharing and planning systems (blog/wikis) used to share sensitive information, such as Intellipedia [28] and SKIWEB [5], have similar needs.

For applications such as these, developers need a methodology by which to enforce rich, fine-grained security policies in an efficient and trustworthy manner. There are two main approaches. A *database-centric* approach relies on native security support provided by the DBMS. For example, Oracle 10g [23] supports a simple form of *row-level* security in which security labels can be stored with individual rows, and the security semantics of these labels is enforced by the DBMS during database accesses. A similar approach is possible with views backed by user-defined functions [23]. A customized row-level security label is hidden by the view, and the label's semantics is transparently enforced by the DBMS via invocations to user-defined functions as part of query processing. For our medical records example, we could apply either approach by mapping portions of a medical record to individual rows, and give each a different label.

Alternatively, a *server-centric* approach is to enforce application-specific policies in the server. For our example, the programmer could define a custom format for access-control labels, store these with rows as above, and then perform access control checks explicitly in the server prior to security-sensitive operations. This is the basic approach taken by J2EE [17] and other application frameworks.

Neither approach is ideal. The database-centric approach is attractive because highly-optimized policy enforcement code is written once for the database for all applications, rather than once per application, improving efficiency and trustworthiness. On the other hand, DBMS support tends to be coarse-grained and/or too specialized. For example, most DBMSs provide only simple access control policies at the table level, and Oracle's relatively sophisticated per-row labels only apply to totally-ordered multi-level security policies [13]. Even customized support based on views, or further native security extensions, will only go so far: some policies simply cannot be enforced entirely within the database. For example, an end-to-end information flow policy [30] requires tracking data flows through the server to ensure, for instance, that the server does not write confidential data to a publicly-viewable web server log.

The server-centric approach has the opposite characteristics: it can enforce highly-expressive application-specific policies, but is potentially far less efficient and less trustworthy. In the worst case the server must load entire database tables into server memory to access and interpret the custom security labels associated with each row. And because the application performs security checks explicitly, programming errors can create security vulnerabilities.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

As a remedy to this state of affairs, we present SELinks, a new programming language for building secure, multi-tier applications that combines the best features of the server-centric and database-centric enforcement strategies. SELinks employs a server-centric programming model for maximum policy expressiveness, and uses compilation and verification techniques to make performance and trustworthiness competitive with the database-centric approach.

SELinks is extension to Links [11], a high-level functional programming language for writing multi-tier web applications (SELinks stands for *security-enhanced* Links). With Links, programmers write a three-tier application as a single program. The Links compiler splits the program into three parts, producing Javascript to run on the browser, SQL to run as queries sent to the database, with the remainder to be run by the Links interpreter on the server. SELinks extends Links to allow programmers to define custom security labels using algebraic and structured types, and to associate these labels with sensitive data. Programmers also define *enforcement policy* functions that govern access to labeled data. These functions are called explicitly by the application as needed. (Section 2.)

To ensure good performance, when calls to enforcement functions occur in queries to the database, the SELinks compiler translates enforcement code to *user-defined functions* (UDFs) stored in the database, and these can be invoked directly during query processing. On the other hand, enforcement functions can also be called as necessary within the server to enforce more expressive, end-to-end policies, e.g., for tracking information flow.

To prevent coding mistakes in which calls to enforcement functions are left out or are performed incorrectly, SELinks extends Links with a novel type system called Fable [34]. The Fable declaration $\mathsf{int}\{l\}$ o ascribes object o a *labeled type*, which indicates in this case that o is an integer protected by object l, a security label. Values of labeled type can only be accessed within enforcement policy functions; the type checker ensures that the main program treats labeled data abstractly, and thus data access cannot bypass enforcement code. Fable support makes SELinks potentially *more* trustworthy than even a database-centric approach, since the DBMS could itself have coding errors.

As a final benefit, SELinks makes secure applications more portable. Security policy enforcement relies only on common DBMS support for user-defined functions, and not on particular security features of the DBMS. Because programmers write enforcement functions in SELinks' high-level language, they need not write variants of their application for different UDF languages. At the moment our implementation (Section 3) targets only PostgreSQL, but we believe other DBMSs could be easily supported.

We have used SELinks to implement two applications that enforce interesting security policies (Section 4). The first is SEWiki, an on-line document management system that allows sensitive documents to be shared securely across a community of users. SEWiki implements a combination of a fine-grained access control policy, a data provenance policy [7], and a declassification policy based on stack inspection [15]. It illustrates SELinks' support for complex label models and highlights our ability to efficiently enforce application-specific policies by compiling enforcement logic to run at the database. We have also implemented SEWineStore, an e-commerce application that implements a fine-grained access control policy. We were able to reuse much of the policy code across the applications, indicating that SELinks promotes the modular enforcement of security policies. Performance experiments (Section 5) show that cross-tier enforcement in SELinks substantially improves application throughput when compared to server-only enforcement.

In summary, the core contribution of this paper is SELinks, a programming language for building multi-tier applications that enforce expressive security policies in an efficient, reliable, and portable manner. The remainder of the paper presents an overview of SELinks, and information about its implementation, application experience, performance, and related work.

2. OVERVIEW

We begin by describing Links [11], the programming language on which SELinks is based. Next we describe, by example, how one builds a secure multi-tier application in SELinks and present a detailed overview of our cross-tier policy enforcement strategy. We conclude by comparing our approach to security in SELinks with popular alternatives and illustrating that no existing approach provides the same combination of efficiency, expressiveness, portability, and trustworthiness as SELinks.

2.1 Links

Modern web applications are often designed using a three-tier architecture. The part of the application related to the user interface runs in a client's web browser. The bulk of the application logic typically runs at a web server. The server, in turn, interacts with a relational database that serves as a high-efficiency persistent store.

Programming such an application can be challenging for a number of reasons. First, the programmer typically must be proficient in a number of different languages—for example, client code may be written as Javascript; server code in a language like Java, C#, or PHP; and data-access code in SQL. Furthermore, the interfaces between the tiers are cumbersome—the data submitted by the client tier (via AJAX [16], or from an HTML form) is not always in a form most suitable for processing at the server or database. These factors are elements of the so-called *impedance mismatch* in web programming.

Links aims to reduce this impedance mismatch by making it easier to synchronize the interaction between the tiers of a web application. The programmer writes a *single* Links program in which client-server communication is via normal function calls, and server queries to the database are expressed as list comprehensions, in the style of Kleisli [37]. The Links compiler compiles client-side functions to Javascript to run in the browser and implements calls from client to server using AJAX. List comprehensions are compiled to SQL expressions that will run on the database. Thus programs are expressed at a fairly high-level while the low-level details are handled by the compiler.

2.2 Building Secure Applications in SELinks

SELinks extends Links with support for enforcing custom security policies. To illustrate how one expresses a security policy in SELinks, we sketch some aspects of the implementation of SEWiki, our secure document-management system. SEWiki is designed to enable secure sharing of documents among users. To accomplish this, we associate secu-

rity labels with fragments of documents and prevent unauthorized access by filtering out confidential content when serving a document to a user. SEWIKI is discussed in greater detail in Section 4.

In SELINKS, implementing a security policy proceeds in three steps. First, we must define the form of *security labels* which are used to denote policies for the application's security-sensitive objects. Second, we must define the *enforcement policy* functions that implement the enforcement semantics for these labels. Finally, we must modify the application so that security-sensitive operations are prefaced with calls to the enforcement policy code. We elaborate on these three steps in the context of SEWIKI.

Security labels. SEWIKI security labels combine policies for access control, provenance, and declassification. Labels are specified using LINKS algebraic datatypes, similar to those present in functional languages like ML and Haskell.

typename *Group* = Principal(*String*) | Auditors | Admins
typename *Acl* = (read:*List*(*Group*), write:*List*(*Group*))
typename *DocLabel* = (acl: *Acl*, prov: *Prov*, declass: *Declass*)

Documents are protected with security labels with the type *DocLabel*, which is a record type with three fields, acl, prov and declass, representing labels from the access control, provenance tracking, and declassification policies, respectively. For now we focus on the access control part. The type *Acl* is itself a record containing two fields, read and write, that maintain the list of groups authorized to read and modify a document, respectively. At the moment, we have three kinds of groups: Principal(x), stands for the group that contains a single user x; Auditors, is the group of users that are authorized to audit a document; and Admins, which include only the system administrators.

Given this label model, the schema for our document database can be written as follows, where each row is given a separate security label:

```
var doc_table_handle = table "documents" with
    (docid : Int,
     doclab : DocLabel,
     text : String{doclab}) from database "docDB";
```

The table contains three columns. The first column is the primary key. The second column stores the row's security label, having type *DocLabel*. The third column's data has labeled type *String*{doclab}, which states that it contains a document element (of type *String*) that is *protected* by the security label stored in the doclab field of the table. This is a kind of *dependent type* [2], and as we discuss later, allows the type system to ensure that this data is not accessed prior to checking the policy.

Enforcement Policy. The next step is to define what labels *mean*, in terms of what actions they permit or deny. The application writer does this by writing special functions, collectively called the *enforcement policy*. The type system ensures that data with a labeled type like *String*{doclab} is treated abstractly by the program. While a *String* can be printed, searched, etc., a *String*{doclab} cannot; its values can never be inspected directly by application code. To access the contents of such data, application code must call an enforcement policy function, passing in the appropriate labels, credentials and application state. Depending on the values passed in, the policy function can choose to either grant or deny access to the data.

For SEWIKI, we implement an authorization check as an enforcement policy function:

```
fun access_str (cred, doclab, x:String{doclab}) policy {
  if (member cred doclab.acl.read) { Some(unlabel(x)) }
  else { None }
}
```

The argument cred is the user's login credential, and has type *Group*. doclab is a *DocLabel* and x is data protected by this label, having type *String*{doclab}. (LINKS type inference infers the types of the first two arguments.) The function returns a value of type *String* option. The LINKS type *t* option is a variant type (a kind of tagged union) consisting either of the value None, or the value Some(x) where x has type *t*. This function is marked with the **policy** qualifier to indicate that it is a part of the enforcement policy.

In the body of the function, we check whether the user's credential is a member of the doclab's read access control list (using the standard member function, not shown). If it is, the user has read privileges on this document, so the policy function uses a special **unlabel** operator to coerce the type of x from *String*{doclab} to *String*, making it accessible once returned from the function. The **unlabel** operator may only appear in enforcement policy code, which ensures labeled data accessible only by way of the enforcement policy.

The combination of dependent types and enforcement policies ensures that programmers always call the appropriate authorization check before manipulating protected data. With a simple formalism called FABLE we have shown that this approach can be used to express and reliably enforce a wide variety of security policies [34]. SELINKS implements FABLE as part of its type checker.

Mediate actions. The final step is to preface security-sensitive accesses to data with calls to enforcement policy functions. Here is a function that performs text search on the document database.

```
1 fun getSearchResults(cred, keyword) server {
2   for(var row ← doc_table_handle)
3   where (var txtOpt = access_str(cred, row.doclab, row.text);
4            switch(txtOpt) {
5            case Some(data) → data ∼ /.*{keyword}.*/
6            case None → false
7            })
8     [row]
9 }
```

The getSearchResults function runs at the server (as evinced by the **server** annotation on the first line), and takes as arguments the user's credential cred and the search phrase keyword. The body of the function is a single list comprehension that selects data from the documents table. In particular, for each row in the table (the syntax **for**(**var** row ← doc_table_handle)) for which the **where** clause is true, the row is included in the final list to which the comprehension evaluates (the syntax [row]). The **where**-clause is not permitted to examine the contents of row.text directly because it has a labeled type *String*{row.doclab}. Therefore, at line 3, we call the access_str policy function, passing in the user's credential, the security label, and the protected text data. If the user is authorized to access the labeled text field of the row, then access_str reveals the data and returns it within a *String* option. Lines 4-7 check the form of txtOpt. If the user has been granted access (the first case), then we check

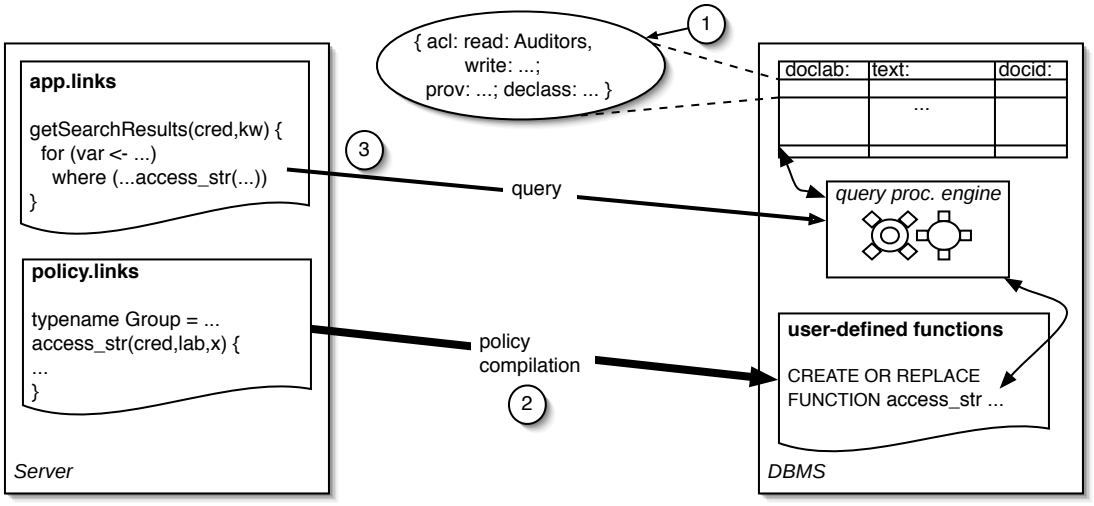

Figure 1: Cross-tier Policy Enforcement in SELinks

if the revealed data matches the regular expression. If the user is not granted access, the keyword search fails and the row is not included.

2.3 Cross-tier Enforcement

Links compiles list comprehensions to SQL queries. Unfortunately, for queries like getSearchResults that contain a call to a Links function, the compiler brings all of the relevant table rows into the server (essentially via the query `SELECT * FROM documents`) so that each can be passed to a call to the local function. This is one of the two main drawbacks of the server-centric approach: enforcing a custom policy may require moving excessive amounts of data to the server to perform the security check there.

SELinks avoids this problem by compiling enforcement policy functions that appear in queries (like access_str) to *user-defined functions* (UDFs) that reside in the database. Queries running at the database can call out to UDFs during query processing, thus avoiding the need to bring all the data to the server. Our implementation currently uses PostgreSQL but should just as well with other DBMSs.

We implement this approach with three extensions to the Links compiler.[1] First, we extend it to support storing complex Links values (most notably, security labels like those of type *DocLabel*) in the database. Prior to this modification, Links only supported storing base types (e.g., integers, floating point numbers, strings, etc.) in database tables. Second, we extend the Links code generator so that enforcement policy functions can be compiled to UDFs and stored in the database. Finally, we extend the Links query compiler to include calls to UDF versions of enforcement policy functions in generated SQL. Each respective step is labeled (1), (2), and (3) in Figure 1.

Representing complex SELinks **data in the database.** The simplest way to encode a Links value of complex type into a database-friendly form would be to convert it to a string. The drawback of doing so is that UDFs would have to either directly manipulate the string encoding or else convert the string to something more usable each time the UDF was called. Therefore, we extend the Links compiler to construct a PostgreSQL *user-defined type* (UDT) for each complex Links type possibly referenced or stored in a UDF or table [25]. To define a UDT, the user provides C-style `struct` declaration to represent the UDT's native representation, a pair of functions for converting to/from this representation and a string, and a series of utility functions for extracting components from a UDT, and for comparing UDT values. UDT values are communicated between the server and the database as strings, but stored and manipulated on the database in the native format. In SELinks, UDTs are produced automatically by the compiler.

At the top of the DBMS tier in Figure 1, we show the three columns that store SEWiki documents. The doclab column depicts storage of a complex *DocLabel* record. This value is compiled to a C struct that represents this label. Section 3.1 discusses our custom datatype support in detail.

Compiling policy code to UDFs. So that enforcement policy functions like access_str can be called during query processing on the database, SELinks compiles them to database-resident UDFs written in PL/pgSQL, a C-like procedural language. (Similar UDF languages are available for other DBMSs.) SELinks extends the Links compiler with a code generator for PL/pgSQL that supports a fairly large subset of the SELinks language; notably, we do not currently support higher-order functions. The generated code uses the UDT definitions produced by the compiler in the first step when producing code to access complex types. For example, Links operations for extracting components of a variant type by pattern matching are translated into the corresponding operations for projecting out fields from C structs. Section 3.2 describes the compilation process.

Figure 1 illustrates that UDFs are compiled from links policy code in the file policy.links. We note that policy code can, if necessary, be called directly by the application program, in file app.links, running at the server.

Compiling Links **queries to SQL.** The final step is to extend the Links list comprehension compiler so that queries like that in getSearchResults can call policy UDFs in the

[1] Our extensions are to revision `r995` in the Links subversion repository (May 2007).

database. This is fairly straightforward. Calls to UDFs that occur in comprehensions are included in the generated SQL, and any LINKS values of complex type are converted to their string representation; these representations will be converted to the native UDT representation in the DBMS. Section 3.3 shows the precise form of the SQL queries produced by our compiler.

2.4 Comparison to Alternative Approaches

To understand the benefits of our approach with SELINKS, we consider some alternative approaches to securing a document-management application.

Database-side enforcement. Some DBMSs aim to enforce a fine-grained policies directly, with little or no application assistance. For example, Oracle 10g [23] has native support for schemas in which each row includes a security label that protects access to that row. In this case, the label model and the enforcement policy are provided directly by the DBMS. As a result, the application code does not need to be trusted to perform the security checks correctly since the DBMS will perform them transparently. Application programmers need only focus on the functional requirements; i.e., they can write queries like (using LINKS syntax):

```
for(var row ← doc_table_handle)
  where (row.text ∼ /.*{keyword}.*/)
    [row]
}
```

Native support for authorization checks in the DBMS can be optimized.

There are two downsides to a database-only enforcement model. The first problem is the lack of customizability. Each DBMS has different security mechanisms, and these may not easily map to application concerns. For instance, Oracle's row-level security is geared primarily to a hierarchical model of security labels, in which security labels are represented by integers that denote privilege levels. A user with privilege at level l_1 may access a row labeled l_2 assuming $l_1 \geq l_2$. While useful, this native support is not sufficient to implement the label model we described above. For one, a typical encoding of access control lists in a hierarchical model requires a lattice model of security [13], rather than the total-order approach used in Oracle. Encoding principal sets in a hierarchical model is also not robust with respect to dynamic policy changes [36]. Furthermore, Oracle 10g is atypical—most DBMSs provide a far more impoverished security model. For instance, PostgreSQL [26], SQLServer [33], and MySQL [21] all provide roughly the same security model, based on discretionary role-based access control [22]. Object privileges are coarse-grained (read, write, execute etc.) and apply at the level of tables, columns, views, or stored procedures. By contrast, SELINKS labels can be defined using LINKS' rich datatype specification facility, labels can be associated with data at varying granularity (table, row, or even within a row), and these labels can be given user-defined semantics via the enforcement policy.

The second problem is that database-only enforcement does not solve the *end-to-end* security problem—while we may be confident that no data moves from the database to the server without proper access, the DBMS cannot ensure the server does not (inadvertently or maliciously) release the data inappropriately, e.g., by writing it to a publicly-visible web log. By contrast, SELINKS ensures that sensitive data, whether accessed via a database query or a server action, is always mediated by a call to the enforcement policy. This provides a level of trustworthiness similar to application-transparent enforcement within the DBMS, but with greater scope. Indeed, it opens up the possibility for enforcing policies that combine information available in the database and the server. In Section 4 we present a novel policy based on Java-style stack inspection [17] that considers the application's call-stack while performing filtering at the database.

Server-side enforcement. Another common approach is to enforce fine-grained security policies primarily in the server. This is the approach taken in the web application frameworks, like J2EE and ASP.NET. In J2EE [17], *Entity Enterprise Java Beans (EJBs)* are used to represent database rows at the server where row data is made available via user-defined methods. For our example we could define a method `findByKeyword` to search a document's text. Access to this method (and other operations) is controlled using the Java Authentication and Authorization Service (JAAS) to invoke user-defined functions under relevant circumstances. ASP.NET is similar to J2EE except it integrates more cleanly with authentication services provided by the Windows operating system [3]. Other lightweight approaches to web programming, like PHP [24] or Ruby On Rails [29], take a more ad hoc approach to security—a set of best practices is recommended to protect applications from common vulnerabilities like code injection attacks. All these approaches are extremely flexible. As with SELINKS, the developer can customize the label model and its semantics. Because policies are enforced at the server, they can consider server and database context, providing broader scope.

The main drawback of the server-side approach is the performance hit that comes with moving data from the database to the server, potentially unnecessarily. As illustration of this, Cecchet et al [8] report that J2EE implementations based on entity beans can be up to an order of magnitude slower than those that do not. That said, Spacco and Pugh [27] report that for the same application much of the performance can be restored with some additional design and tuning, but this can be a frustrating and brittle process. The other problem with server-side enforcement is trustworthiness: the application programmer is responsible for correctly invoking security policy functions manually, so that mistakes can lead to security vulnerabilities.

Hybrid enforcement. SELINKS essentially represents a kind of hybrid enforcement strategy: it presents a server-side programming model but compiles server functions to UDFs to allow them to run on the database and thus optimize performance. This same basic strategy could also be encoded "by hand." One could define a custom notion of security label (e.g., as a certain format of strings), and then write a series of user-defined functions akin to the SELINKS enforcement policy for interpreting these strings. The application writer would then be responsible for calling these functions during database accesses to enforce security. To avoid changing the application, a popular alternative is to have the DBMS perform UDF calls transparently when accessing the database via a *view* [23]. For example, we could define a view of our document table as containing only the docid and text fields; when querying these fields, calls to UDFs would be made by the DBMS transparently to filter results according to the hidden doclab field.

This by-hand approach has three main drawbacks, compared to what is provided by SELinks. First, database-resident functions are painfully low-level, operating on application object *encodings* rather than, as in SELinks, the objects themselves. Second, different DBMSs have different UDF languages, and thus a manual approach requires possibly many implementations; by contrast the SELinks compiler can be used to target many possible UDF languages. Finally, if application programmers must construct queries with the appropriate calls to security enforcement functions there is the danger that coding errors could result in a policy being circumvented. While using views reduces the likelihood of this problem, there are still parts of the application that manage the policy, e.g., by updating the doclab portions of the objects, and these bits of code are subject to mistakes. The SELinks type checker ensures that operations on sensitive data (whether in queries like our keyword search example or in operations such as server logging functions) respect the security policy.

3. IMPLEMENTATION OF SELINKS

In this section, we present the details of the cross-tier policy-enforcement features of the compiler, overviewed in Section 2.3. We describe our data model for storing SELinks values in PostgreSQL using user-defined types, illustrate how we compile SELinks functions to user-defined functions, and explain how we compile SELinks queries to make use of these functions and manipulate complex SELinks data. Details of the Fable extensions to the Links type system have been discussed elsewhere [34].

3.1 User-defined Type Extensions in PostgreSQL

User-defined types (UDTs) in PostgreSQL are created by writing a shared library in C and dynamically linking it with the database. For each UDT, the library must define three things: an in-memory representation of the type, conversion routines to and from a textual representation of the type, and functions for examining UDT values. Our in-memory representation for SELinks values is centered around the `Value`, `Variant`, and `Record` structures, shown in Fig. 2.

The `Value` type defines a variable-length data structure that represents all SELinks values. The first field vl_len_ (used by all the structures) is used to store the size (in memory words) of the represented SELinks value. The remainder of the structure defines a tagged union: the field type is a tag denoting the specific variant of the value field that follows. All the possible forms of SELinks values are recorded in the value union, including variants (like *Group*), records (like *Acl*), integers, and strings.

The `Variant` type represents an SELinks value that inhabits a variant type. Every instance of a `Variant` type consists of a single *constructor* applied to a `Value` (stored in the value field of the `Variant` structure). For example, a SELinks value like Principal(''Alice'') is represented in the database as an object of type `Variant` where the label field contains the zero-terminated string ''Principal'', and the value field is a `Value` whose type field indicates it is a string, with the string's value stored in the string field of the value union.

The `Record` type represents a record that can hold an arbitrary number of SELinks values of different types. In particular, it is used to store the values of multi-argument labels; for example, ActsFor(''Alice'',''Bob'') is a `Variant` whose value is `Record` ("Alice", "Bob"). A record's field names are omitted (the name is implied by position).

```
typedef struct Value {
  int32 vl_len_;
  int32 type;
  union {
    Variant variant;
    Record record;
    int32 integer;
    text string;
    ...
  } value;
} Value;

typedef struct Variant {
  int32 vl_len_;
  char* label;
  Value value;
} Variant;

typedef struct Record {
  int32 vl_len_;
  int32 num_args;
  Value value;
  Record rest;
} Record;

Variant* variant_in(cstring);
cstring variant_out(Variant*);
boolean variant_eq(Variant*, Variant*);

Variant* variant_init(text, anyelement);
text variant_get_label(Variant*);
Record* variant_get_record(Variant*);
Variant* variant_get_variant(Variant*);
int32 variant_get_integer(Variant*);
text variant_get_string(Variant*);

Record* record_in(cstring);
cstring record_out(Record*);

Record* record_init(anyelement);
Record* record_set(Record*, int32, anyelement);
text record_get_string(Record*, int32);
```

Figure 2: PostgreSQL User-Defined Types

Some of the functions which work on these data types are listed in Fig. 2. The string conversion functions end with the suffixes _in and _out. These are used internally by PostgreSQL to translate between a UDT's in-memory and string representation. Since our composite types allow embedded values, the *_in functions must be able to recursively parse subexpressions (e.g., in "Principal(`"Alice"`)", the `"Alice"` subexpression must be parsed as a string).

The variant_eq function compares two `Variant` types for equality; in PostgreSQL, it is called by overloading the "=" operator. The variant_eq function implements a special pattern matching syntax, where the value "_" is treated a wild card, and will match any subexpression. For example, $Acl(\texttt{"Alice"}) = Acl(\_)$ is true.

The variant_get_label function returns the text label of a `Variant`, while the variant_get_* functions get the value of the `Variant`; if the type does not match, a run-time error occurs. We require a different accessor function for each type because PostgreSQL requires return variables to have a type. On the other hand, the variant_init function, which creates a new `Variant` type, takes an argument of type **anyelement**. This is a PostgreSQL "pseudo-type" that accepts any type of argument; the actual type can be determined dynamically. This allows us to create user-defined functions that take a polymorphic type (such as access, described in the next section).

The `Record` functions are similar to the `Variant` functions.

```
1.  CREATE FUNCTION access(text,record,anyelement)
2.  RETURNS variant AS $$
3.  DECLARE
4.    cred   ALIAS FOR $1;
5.    doclab ALIAS FOR $2;
6.    x      ALIAS FOR $3;
7.  BEGIN
8.    IF member(cred,record_get_rec(
                record_get_rec(doclab, 0),0)) THEN
9.      RETURN variant_init('Some', x);
10.   ELSE
11.     RETURN 'None';
12.   END IF;
13. END;
14  $$ language 'plpgsql'
```

Figure 3: Generated PL/pgSQL code for access

```
1. SELECT docid, doclab, text FROM
2.  (SELECT
3.     S.doclab as doclab,  S.docid as docid,
4.     S.text as text,
5.     access('Alice', S.doclab, S.text) AS tmp1,
6.   FROM documents AS S
7.   ) as T
8. WHERE
9.   CASE
10.     WHEN ((T.tmp1 = 'Some((_))'))
11.      THEN (variant_get_str(T.tmp1)
              LIKE '%keyword%')
12.    WHEN (true)
13.      THEN false
14.  END
```

Figure 4: SQL query generated for getSearchResults

The record_get_$*$ functions take a record x and a (zero-based) integer index i as arguments and returns the ith component of the record x, if such a component exists and is of the proper type. If either condition is unsatisfied, then a run-time error results. record_init creates a new single record with the given value, while record_set sets a record's value, possibly extending the record by one element as a result.

In the remainder of this section we show how these types are used both within our compiled UDFs as well as in the body of SQL queries.

3.2 Compilation of SELinks to PL/pgSQL

To compile SELinks functions to UDFs, we built a new Links code generator that produces PL/pgSQL code, one of PostgreSQL's various UDF languages. Prior to our extension the Links code generator could only generate Javascript code for running on the client. PostgreSQL supports several different UDF languages, but PL/pgSQL is the most-widely used. It has has a C-like syntax and is fairly close to Oracle's PL/SQL.[2]

Code generation is straightforward, so we simply show an example. Figure 3 shows the (slightly simplified) code generated for an enforcement policy function called access, a generalization of the function access_str shown in Section 2.2, that can take any type of argument (which is useful when labels annotate values of many different types, since we can write a single access function rather than one per type). A function definition in PL/pgSQL begins with a declaration of the function's name and the types of its arguments. Thus, line 1 of Figure 3 defines a UDF called `access` that takes three arguments of built-in type `text`, a custom type `record`, and the special built-in "pseudo-type" `anyelement`. The `anyelement` type allows us to (relatively faithfully) translate usages of polymorphic types (as in the argument of our generalized access function) in SELinks to PL/pgSQL. At line 2, we define the return type of `access` to be `variant`, since it is supposed to return an option type.

At lines 4, 5, and 6, we give names to the positional parameters of the function by using the `ALIAS` command (a peculiarity of PostgreSQL). That is, the first argument is named `cred` to represent the credential; the second argument is `doclab` to represent the security label of *DocLabel* type; the final argument `x`, is protected data of any type.

In the body of the function, lines 8-12, we check if the user's credential `cred` is mentioned in the doclab.acl.read field. Accessing this field requires first projecting out the record doclab.read, using `record_get_rec(doclab, 0)` and then the read field using a similar construction. The authorization check at line 8 relies on another UDF (`member`) whose definition is not shown here.

If this authorization check succeeds, at line 9 we return a value corresponding to the SELinks value Some(x). Notice that the **unlabel** operator that appears in SELinks is simply erased in PL/pgSQL—it has no run-time significance. If the check fails, at line 10 we return the nullary variant construct None.

3.3 Invoking UDFs in Queries

The last element of our cross-tier enforcement strategy is to compile SELinks comprehension queries to SQL queries that can include calls to the appropriate policy UDFs. This is built on infrastructure provided by Dubochet [14] (based on work in Kleisli [37]). Prior to our extensions, the Links compiler was only capable of handling relatively simple queries. For instance, queries like our keyword search with function calls and case-analysis constructs were not supported.

Figure 4 shows the SQL generated by our compiler for the keyword search query in the body of getSearchResults. This query uses a sub-query to invoke the `access` policy UDF and filters the result based on the value returned by the authorization check. We start by describing the sub-query on lines 2–5. Lines 3 and 4 select the relevant columns from the `documents` table; line 5 calls the policy function `access`, passing in as arguments the user credential (here, just the username `'Alice'`, but, in practice, an unforgeable authentication token); the document label field `S.doclab`; and the protected text `S.text`, respectively. The result of the authorization check is named `tmp1` in the sub-query.

Next, we describe the structure of the where-clause in the main query, at lines 8–14. We examine the the value returned by the authorization check; if we have obtained a Some(x) value, then we search x to see if it contains the keyword, otherwise the where-clause fails. Thus, at line 10, we check that `T.tmp1`, the result of the authorization check

[2]Note that, unlike most database systems, PostgreSQL makes no distinction between stored procedures and user-defined functions.

for this row, matches the special variant pattern `Some((_))`. In this case, the test on line 10 is satisfied if the value `T.tmp1` is the variant constructor `Some` applied to *any argument*. If this pattern matching succeeds, at line 11, we project out the string argument of variant constructor using the function `variant_get_str`. Once we have projected out the text of the document, we can test to see if it contains the keyword using SQL's `LIKE` operator. Lines 12–13 handle the case where the authorization check fails.

Finally, we turn to line 1 of this query which selects only a subset of the columns produced by the sub-query. The reason is efficiency: we do not wish to pass the temporary results of the authorization checks (the `T.tmp1` field) when returning a result set to the server.

Although our code generators are fairly powerful, there are some features that are not currently supported. First, our current label model requires storing a security label within the same row as the data that it protects. Next, our support for complex join queries as well as table updates is still primitive. We anticipate improving our implementations to handle these features in the near future. Finally, as mentioned earlier, we do not allow function closures to be passed from server to the database; however, we do not foresee this being a severe restriction in the short term.

4. APPLICATION EXPERIENCE

This section illustrates that SELinks can support applications that enforce of fine-grained, custom security policies. We present two examples we have developed, a blog/wiki SEWiki, and an on-line store SEWineStore. Demos of both applications can be found at the SELinks website, `http://www.cs.umd.edu/projects/PL/selinks`.

4.1 SEWiki

SEWiki is an on-line document-management system inspired by *Intellipedia* [28] and SKIWEB [5], which are blog/wiki-style applications designed to promote information sharing among U.S. intelligence agencies and the Dept. of Defense, respectively. We begin by discussing several requirements of such an application and then present the salient features of our implementation.

4.1.1 SEWiki Requirements

SEWiki aims to satisfy three requirements:

Requirement 1: Fine-grained secure sharing. SEWiki aims to maximize the sharing of critical information across a broad community without compromising its security. To do this, SEWiki enforces security policies on *fragments of a document*, not just on entire documents. This allows certain sections of a document to be accessible to some principals but not others. For example, the source of sensitive information may be considered to be high-security, visible to only a few, but the information itself may be made more broadly available.

Requirement 2: Information integrity assurance. More liberal and rapid information sharing increases the risk of harm. To mitigate that harm, SEWiki aims to ensure the integrity of information, and also to track its history, from the original sources through various revisions. This permits assessments of the quality of information and audits that can assign blame when information is leaked or degraded.

Requirement 3: Controlled release of secret/private information. Every information protection policy must account for exceptional cases. Sensitive information that is usually withheld from less privileged principals may, in special circumstances, have to be released. For example, financial information may be kept private until a person's death, but then it is revealed to the next of kin. Declassification typically is governed by policies that dictate *when* information released (e.g., after a person's death), in *what* form (e.g., with certain personal information removed), *who* may view the released information (e.g., next of kin), and *where* (in the program) the release may take place [31].

As discussed in the introduction, these requirements are germane to a wide variety of information systems, such as on-line medical information systems, e-voting applications, and on-line stores.

4.1.2 SEWiki Implementation

SEWiki consists of approximately 3500 lines of SELinks code. It enforces a combined group-based access control policy, provenance policy, and stack inspection-based declassification policy. Policies are expressed as security labels having type *DocLabel*, a record type we introduced in Section 2.2 that has three fields, acl, prov, and declass, which represent the access control, provenance, and declassification policies, respectively:

typename *DocLabel* = (acl: *Acl*, prov: *Prov*, declass: *Declass*)

SEWiki label-based policies can be applied at a fine granularity. In what follows we discuss SEWiki's document model and the three policy elements of a *DocLabel*.

Document structure. An SEWiki document is represented as a tree, where each node represents a security-relevant section of a document at an arbitrary granularity—a paragraph, a sentence, or even a word. Security labels are associated with each node in the tree. When manipulating documents within the server, the document data structure is implemented as a variant type. To store these trees in the relational database, we extend the schema representation presented in Section 2.2 with three new fields as follows:

```
var doc_table_handle_full = table "documents" with
  (docid : Int, doclab : DocLabel, text : String{doclab},
   parentid: Int, sibling: Int, ischild: Bool
) from database "docDB";
```

The parentid field is a foreign key to the docid of the node's parent, the sibling field is an index used to display the sub-documents in sequential order, and the ischild field is used to indicate whether this node is a leaf (containing text) or a structural node (containing sub nodes). To retrieve an entire document, we fetch the parent, look up all the immediate children (by searching for nodes with a parentid of the parent), then recursively look up all the children's children, until we retrieve all the leaf nodes.

The policy label of a parent node indirectly and uniformly restricts access to all its children; labels that appear at child nodes may add to this restriction. To implement this tree-based semantics literally would require recursive policy checks, which is expensive. Instead, we allow only the local node's label to be checked by ensuring it always reflects the policies of its ancestors' labels. In particular, whenever a parent node's policy is changed to add further restriction (e.g., by adding a *Group* value to the read access control list), this addition is propagated to all the labels of the children

nodes as well. On the other hand, relaxing a policy at the parent (e.g., by removing an element from the read list) is *not* reflected to the children automatically—we felt this was a more sensible choice of "user interface" to policy changes, which reflects future changes to the parent document, while leaving the policies on the children the same as before.

An important benefit of our approach is that it enables an application to make design choices that are pertinent to security, and have them reliably enforced in the server using the abstractions that are available there. In this case, enforcing our recursive policy checks in the database would be both cumbersome and inefficient because SQL is not particularly well suited to handling recursive relations. Thus, even in situations where end-to-end tracking of information flow is not essential (as in our access control policy here) being able to address security-related concerns in the server is crucial.

Access control policies. As discussed in Section 2.2, access control labels (of type *Acl*) consist of separate lists for read- and write-permissions. These address our confidentiality and integrity (tamper prevention) requirements.

Data provenance tracking. We also address information integrity by maintaining a precise revision history in the labels of each document node—this is a form of data provenance tracking [7]. This part of labels, having type *Prov*, is defined as follows:

typename *Op* = Create | Edit | Del | Restore | Copy | Relab
typename *Prov* = *List*(oper:*Op*, user:*String*, time:*String*)

A provenance label of a document node consists of a list of operations performed on that node together with the identity of the user that authorized that operation and a time stamp. Tracked operations are of type *Op* and include document creation, modification, deletion and restoration (documents are never completely deleted in SEWiki), copy-pasting from other documents, and document relabeling. For the last, authorized users are presented with an interface to alter the access control labels that protect a document.

This provenance model exploits SELinks' support for custom label formats. It is hard to conceive of encoding such a complex label format using some form of native database support for row-level security. Finally, this policy does not directly attempt to protect the provenance data itself from insecure usage. We have shown elsewhere that protecting provenance data is an important concern and is achievable in SELinks without too much difficulty [34, 12].

Stack inspection for declassification. In SEWiki, information release policies—a.k.a. *declassification* policies—are governed by stack inspection [15]. The idea is that certain information should be only released under particular circumstances, and these circumstances are determined by the call stack. Stack inspection policies are in widespread use in mainstream language environments like Java or .NET [17, 6], but here these policies are limited to controlling access to server resources, not resources in the database. For example, stack inspection is used to ensure that untrusted code never directly accesses the server's file system. This may be because tracking the current call-stack and communicating its contents to the database is cumbersome.

In contrast, SELinks can reliably track the state of the current call stack and easily pass a representation of this stack to the database. Some of our recent work explores making call-stack tracking tamper-proof [35] using a Fable-like type system. Here we show how to easily pass that call-stack representation in queries for in-database enforcement.

```
1 fun access_si (stack, cred, doclab, x:String{doclab}) policy {
2     if ((member cred doclab.acl.read) &&
3         (check_fns_on_stack (cred, stack, doclab.declass))) {
4         Some(unlabel(x))
5     } else { None }
6 }
```

Figure 5: Enforcement policy for stack inspection

The downgrading policy of a *DocLabel* has type *Declass*, defined as follows:

typename *Declass* = *List*(user:*Group*, authFun:*String*)

The idea here is that if a user appears in user field of a declassification label (and appears in the normal access control label's read-list) then the user is allowed to view the labeled data only if the given authFun is on the stack, which has the effect of conditionally releasing the information to the user (in this case, according to the *where* dimension of declassification [31]).

For example, suppose we had a document in two parts: one part contains some analysis information and the other contains the source of that information. The *DocLabel* for the analysis portion might look like

(acl: (read:[Principal("Alice")], write:[]), prov:..., declass:[])

This says that a program acting for Alice may read the data under any circumstance. Now suppose the *DocLabel* for the source portion is

```
(acl: (read:[Principal("Alice")], write:[]),
 prov:...,
 declass:[(Principal("Alice"), authFunction="analyzeSrc")])
```

In this case, it says that the program may read the source information on Alice's behalf, *but only if the analyzeSrc function is on the stack*. This might be because this portion of the program only makes use of this information in a certain, acceptable way.

Figure 5 shows the enforcement policy code for our stack inspection policy. It adds an additional argument stack, representing the current call stack, to our earlier access_str function (Section 2.2). Here, the stack is represented as a list of *String*s, one per function name on the stack. The implementation of the function is the same as before, but for the additional call (line 3) to check_fns_on_stack. This function returns true if cred does not appear in the declass portion of x's label (and thus is not governed by a declassification policy), or if it does, that the appropriate authFun appears on the stack.

The interesting thing is that we can enforce this policy on the database rather than on the server. To do this, we compile our policy functions and check_fns_on_stack to user-defined functions and store them in the database. Queries contain authorization checks as before, but this time pass the current stack as an argument

```
for (var row ← doc_table_handle)
where (... access_si(cred, stack, row.label, row.text) ...)
 [row]
```

The SELinks compiler seamlessly translates the stack object (along with the others in the call) into a form that can be passed to the database and used by UDFs stored there.

Other forms of downgrading policies are also easily enforced across tiers in SELinks. For instance, we could imagine a label model in which sensitive data is labeled with filter functions that must be applied to the data when it is selected out of the database. Enforcing such a policy by means of user-defined functions would be straightforward.

4.2 SEWineStore

We extended the "wine store" e-commerce application distributed with Links with security features. We defined labels to represent users and associate these labels with orders, in the shopping cart and in the order history. This helps ensure that an order is only accessed by the customer who created it.

Order information in SEWineStore is represented with the following type:

typename Order = (acl:*Acl*, items:*List*(CartItem){acl})

An order is represented by a record with two fields. The acl field stores a security label while the items field contains the items in the shopping cart. The *Acl* type is the same as that used in SEWiki, and many of the enforcement policy functions are shared between the two applications. In general, we found that access control policies were easy to define and to use, with policy code consisting of roughly 200 lines of code total (including helper functions). Our experience also indicates that it is possible for security experts to carefully program policy code once, and for several applications to benefit from high-reliability security enforcement through policy-code reuse.

5. EXPERIMENTAL RESULTS

In this section, we present the results of an experiment conducted to compare the efficiency of server-side versus database-side policy enforcement. We also examine other factors that come into play when running database applications, such as the number of rows being processed by the query, and the location of the database (local host or network). We also benchmark SELinks against a simple access control program written in C.

We show that, in the case of SELinks, running a policy on the database greatly reduces the total running time compared to running the same policy on the server when tables are large (up to a 15× speed up). In addition, our C implementation highlights the high current overhead of SELinks programs, while at the same time showing that our PostgreSQL implementation is comparable in speed (and shows a slight improvement when network latency is considered).

5.1 Configuration

Our system configuration is shown in Table 1. We ran two different system configurations: a single-server mode (local) where the server and database reside on the same machine (Machine A), and a networked version where the server runs on Machine B.

For our test, we used the getSearchResults query presented in Fig. 4, which checks if a user has access to a record and, if so, returns the record if it contains a particular keyword. We generated two tables of random records (1,000 and 100,000 records), each comprised of 5–20 words selected from a standard corpus. Each record has a 10% probability of containing our keyword, and each record is labeled by a random access control label, which grants access 50% of the time. Thus, the query should return approximately 50 results and 5,000 results for the 1,000-record and 100,000-record tables, respectively.

	Machine A	**Machine B**
CPU:	Intel Quad Core Xeon	
	2.66 GHz	2.0 GHz
RAM:	4.0 GB	2.0 GB
HDD:	7,200 RPM SATA	7,200 RPM EIDE
Network:	100 Mbit/s Ethernet	
OS:	Red Hat Enterprise Linux AS 4	
	Linux kernel 2.6.9	
DBMS:	PostgreSQL 8.2.1	N/A

Table 1: Test platform summary

In running our tests on SELinks, we varied the number of records in the table (1,000 or 100,000), whether the policy was enforced on the server or the database, and the locality of the server (e.g., same or networked machine). We also created a C program that queries the database, manually performs the access control check, and searches for the keyword. The C program operates in one of three modes; no access control, server-side access control, or database-side access control, using the same SQL query as generated SELinks program, including the database-level UDF function.We compare this program against our SELinks implementation for all the tests above. All running times are the mean of five runs.

5.2 Results

The results of our experiment are summarized in Figures 6 and 7, which illustrate the time required to run the query on 1,000 and 100,000 records, respectively. The horizontal axis illustrates the language (C or SELinks) and policy enforcement location (None, Database, or Server) used. For each language/policy pair, we show two bars representing the local or networked database configurations, respectively.

The highlight of both figures is the significant improvement shown in running an SELinks policy on the database rather than the server. For the 100K-record example running over the network we see a 16× improvement; for the 100K-record case with a local database the improvement is 7.5×; and for local and network queries on 1,000 records the improvement is 4×.

The current incarnation of SELinks, however, is an interpreter language with few optimizations. Our C program results illustrate some more general results with regard to this technique. Consider the 100K-record results given in Fig. 7. First, running our C program with no policy enforcement takes a little over one second; this gives us a baseline for how long it takes to retrieve the full data set; this illustrates time that could possibly be saved by reducing the result set at the database. Our C implementation of server access control is *much* faster than our SELinks implementation (≈ 12.5×); this illustrates the lack of optimization in SELinks. That said, the SELinks database policy implementation is comparable to the C version on a single machine, (only 27% slower) and is marginally faster when network transmission is taken into account. It is interesting to note

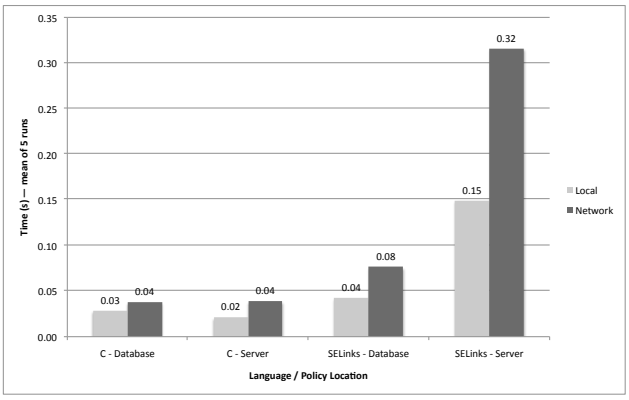

Figure 6: Keyword search on 1,000 rows

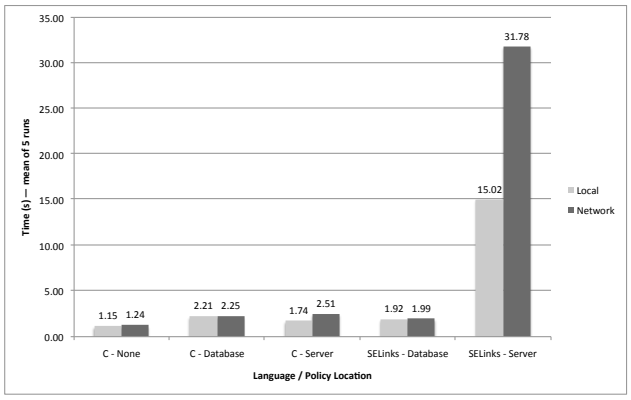

Figure 7: Keyword search on 100,000 rows

that, when running the database-policy versions, the SELinks implementation actually slightly out-performs the C implementation; this indicates that the C implementation may not be as optimized as possible.

In summary, running SELinks policies on the database instead of the server greatly improves performance, particularly for large queries. Based on the comparison with C, we note that the SELinks server component could benefit greatly from more optimization, while database-side enforcement is quite efficient.

6. RELATED WORK

SELinks expands on the original goal of Links [11], which is to reduce the impedance mismatch in programming multi-tier web applications. Our work aims to reduce the impedance mismatch faced when synchronizing the security mechanisms available in DBMSs and server-side programming frameworks. Several other languages also aim to simplify web programming. Hop [32] and the Google Web Toolkit (GWT) [18] both resemble Links in that they provide a unified model for programming web-based client/server applications. Unlike Links, neither Hop nor GWT address the integration of the server and database—but, research on this problem has a long tradition, ranging from Albano et al. [1] to more recent work on Cω [4]. None of this work provides support for enforcing security policies.

Swift [9] and SIF [10] are both high-level languages for programming web applications with security concerns, with a focus on enforcing end-to-end information flow policies. Both languages essentially ignore the database tier (SIF focuses on servlet interactions and Swift considers client-server interactions), while server-database interactions is the focus of our work with SELinks. We plan to consider client-server interactions in future work.

The security models of several mainstream DBMSs and application frameworks were reviewed in Section 2.4. We pointed out that pure server-side approaches can be inefficient and untrustworthy, while a pure database-centric approach is more limited in scope, particularly because end-to-end policies require tracking sensitive operations throughout the application. In this context, SEPostgreSQL [19] is worth special mention. SEPostgreSQL is an extension of PostgreSQL that aims to achieve end-to-end security through integration with the SELinux secure operating system [20]. SEPostgreSQL allows SELinux policy metadata to be associated with tables, columns, and rows in the database. Access to protected objects in SEPostgreSQL is mediated by the SELinux operating system's underlying reference monitor. This opens the possibility of enforcing a uniform policy throughout the operating system and database. However, it is unclear if the ideas behind SEPostgreSQL translate well to other DBMSs. In contrast, a key benefit of our work is portability. We rely only on widely-used features of PostgreSQL (user-defined types and functions) which are also available in most other mainstream DBMSs. The assurance using Fable types to avoid security bypasses is also unique to our approach.

7. CONCLUSIONS

This paper has presented SELinks, a system that allows a database and web server to collaboratively enforce a unified security policy. We have argued that such a multi-tier approach to security is necessary for expressing rich application policies while maintaining efficiency and trustworthiness. We have shown how SELinks can model and enforce a variety of secure application policies, and have described how SELinks implements such policies in the database, via expressive user-defined types, compiling policy functions to user-defined functions, and rewriting SQL queries. We have implemented a wiki application that demonstrates multiple security properties, and have extended an existing Links application by reusing some of the same policies. Finally, we have shown that enforcing policies in the database, versus in the server, improves throughput in SELinks by as much as an order of magnitude.

8. REFERENCES

[1] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[2] D. Aspinall and M. Hoffmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT Press, 2004.

[3] Authentication in ASP.NET: .NET Security Guidance. `http://msdn2.microsoft.com/en-us/library/ms978378.aspx`.

[4] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in cω: The power is in the dot! In *ECOOP '02*, 2002.

[5] R. Boland. Network centricity requires more than circuits and wires. *SIGNAL*, Sept. 2006.

[6] D. Box. *Essential .NET Volume I: The Common Language Runtime.* Addison Wesley, 2002.

[7] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261, New York, NY, USA, 2002. ACM.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07*. ACM Press, 2007.

[10] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security '07*, 2007.

[11] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO '06*, 2006.

[12] B. Corcoran, N. Swamy, and M. Hicks. Combining provenance and security policies in a web-based document management system. In *On-line Proceedings of the Workshop on Principles of Provenance (PrOPr)*, Nov. 2007.

[13] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[14] G. Dubochet. The SLinks Language. Technical report, University of Edinburgh, School of Informatics, 2005.

[15] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *POPL '02*. ACM Press, 2002.

[16] J. J. Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, feb 2005.

[17] L. Gong. *Inside Java 2 platform security architecture, API design, and implementation*. Addison-Wesley, 1999.

[18] Google Web Toolkit. `http://code.google.com/webtoolkit/`.

[19] K. Kohei. The Security-Enhanced PostgreSQL security guide, 2006. Available at `http://code.google.com/p/sepgsql/`.

[20] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *FREENIX 2001*. USENIX Association, 2001.

[21] Security privileges provided by MySQL. `http://dev.mysql.com/doc/refman/5.1/en/privileges-provided.html`.

[22] National Institute of Standards and Technology : Role-based access control. `http://csrc.nist.gov/rbac/`.

[23] Oracle Corporation. Oracle 10g release documentation, 2007. Available at `http://www.oracle.com/technology/documentation/database10g.html`.

[24] PHP Security Consortium: PHP Security Guide. `http://phpsec.org/projects/guide/`.

[25] PostgreSQL Global Development Group. Postgresql 8.2.1 software release, 2007. Available at `http://www.postgresql.org`.

[26] Security privileges provided by PostgreSQL. `http://www.postgresql.org/docs/8.2/static/ddl-priv.html`.

[27] B. Pugh and J. Spacco. Rubis revisited: why j2ee benchmarking is hard. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 204–205, New York, NY, USA, 2004. ACM.

[28] Reuters, October 2006. U.S. Intelligence Unveils Spy Version of Wikipedia.

[29] Securing Your Rails Application. `http://manuals.rubyonrails.com/read/book/8`.

[30] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, Jan. 2003.

[31] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05*. IEEE Computer Society, 2005.

[32] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 975–985, New York, NY, USA, 2006. ACM.

[33] Authorization and permissions in SQLServer. `http://msdn2.microsoft.com/en-us/library/bb669084.aspx`.

[34] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2008.

[35] N. Swamy and M. Hicks. Verified enforcement of automaton-based information release policies. Technical Report CS-TR-4906, Department of Computer Science, University of Maryland, 2008.

[36] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proc. Computer Security Foundations Workshop (CSFW)*, pages 202–216, July 2006.

[37] L. Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1), 2000.