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

Brian J. Corcoran  Michael Hicks  Nikhil Swamy∗
University of Maryland, College Park  Microsoft Research
(bjc,mwh)@cs.umd.edu  nswamy@microsoft.com

ABSTRACT
This paper presents SELINKS, a programming language focused on building secure multi-tier web applications. SELINKS provides a uniform programming model, in the style of LINQ and Ruby on Rails, with language syntax for accessing objects residing either on the database or at the server. Object-level security policies are expressed as labels, which are themselves fully-customizable, first-class objects. Access to labeled data is mediated via trusted, user-provided policy enforcement functions.

SELinks has two novel features that ensure security policies are enforced correctly and efficiently. First, SELINKS uses a novel type system that allows a protected object’s type to refer to its protecting label. This way, the type system can check that labeled data is never accessed directly by the program without first passing through the appropriate policy enforcement function. Second, SELINKS compiles policy enforcement code to database-resident user-defined functions, which can be called directly during query processing. Database-side checking avoids transferring data to the server needlessly, while still allowing policies to be expressed in a customizable and portable manner.

Our experience with two sizeable web applications, a model health-care database and a secure wiki with fine-grained security policies, indicates that cross-tier policy enforcement in SELINKS is flexible, relatively easy to use, and, when compared to a single-tier approach, improves throughput by nearly an order of magnitude.

1. INTRODUCTION
The rise of the web has coincided with the rise of multi-tier applications. These applications consist, at the least, of three tiers: a database for storing persistent data (e.g., inventories, purchase records), a server for handling the application logic (e.g., transaction processing, search, making recommendations), and a client-side web browser for the user interface, which displays results and collects information. Traditionally, each tier was written in its own language or framework; e.g., SQL for query processing on the database, PHP or Java for the server logic, and Javascript and HTML for the client. These different programming languages and tier models create an impedance mismatch that complicates programming multi-tier applications.

Many frameworks have been developed to mitigate the impedance mismatch and provide a more uniform programming model. For example, the LINKS [10] programming language can be used to write a single program that the compiler splits to run at the database, server, and client tiers, seamlessly introducing remote procedure calls and object representation mappings as needed. Similar support is provided by other languages or frameworks, including Microsoft’s Volta [34], the Google Web Toolkit [14], and Hop [15] (unifying the client-server view), and LINQ [18, 17], Java EE (formerly, J2EE) [16], and Ruby on Rails [26] (unifying the server-database view).

Our focus in this paper is to explore how to build applications with security concerns, e.g., because they handle private or sensitive data. A uniform server-database programming model is already a good start. One can naturally enforce security policies directly in terms of application objects, rather than lower-level representations of such objects in the database. Moreover, the application can use its own notions of authority, identity, policy, etc. rather than rely on particular, potentially non-portable (and sometimes inefficient) mechanisms native to a particular DBMS.

On the other hand, there are two key disadvantages to what amounts to a server-side enforcement strategy. First, it can lead to poor performance. When an object’s security policy is expressed as metadata also stored in the database, the server must process a query by transferring and deserializing all potentially-relevant objects so that it can perform the security check (and thus further filter the results). Compared to performing access checks on the DBMS itself, this approach can lead to much higher query latencies and greater network contention if only a small fraction of the transferred objects turn out to be securely accessible. Second, as applications become larger and more complicated, programmers are more likely to make security-relevant coding mistakes, such as missing an access control check on a rarely-taken path. If several applications wish to access the same back-end database, these security checks must be repeated in each application, which increases the chances for mistakes and adds to the programming burden.
This paper presents Security-enhanced Links (or SE-LINKS), an extended version of the LINKS multi-tier programming language with novel support for fine-grained security policy enforcement. SE-LINKS retains the security benefits of the uniform model of multi-tier programming while eliminating its drawbacks. Programmers define security-relevant metadata (termed labels) using algebraic and structured types, and define enforcement policy functions that the application calls explicitly to mediate access to labeled data. While this basic strategy can be implemented in any server-database-uniform model (e.g., LINQ or J2EE), SE-LINKS includes two novel features that prevent coding errors and improve performance.

To ensure that calls to enforcement functions are never left out or performed incorrectly, SE-LINKS implements a novel type system called FABLE [33]. The FABLE declaration `int | 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 are opaque to the main program; e.g., while `int` values can be added, printed, etc. there are no native operations on `int | o` values. To use a labeled object, the main program is thus forced to pass it to an enforcement policy function of the appropriate type. For example, we could pass our labeled integer to the policy function `access_int(;lab, data;int | o, user;cred)`, whose first argument is a label, whose second argument is an integer labeled by that label, and whose third argument is a user credential. The `access_int` function will return the second argument as a normal `int` so long as `user` is permitted access to it, according to `l`. By making the types of enforcement policy functions suitably specific, we can essentially ensure complete mediation: a particular labeled datum must be passed through a particular policy function if it is ever to be used.

To ensure good performance, when calls to enforcement functions occur in database queries, the SE-LINKS compiler translates the calls to user-defined functions (UDFs) accessible during query processing. For example, we have implemented a simple web-based application called SESPIKE, inspired by The Spine [21], which provides access to a patient’s medical records. Each record, stored as a database row, has a corresponding label that encodes an access control list (ACL); the appropriate enforcement policy function is automatically compiled to a UDF and invoked during query processing.

This approach is far more efficient than server-side enforcement but retains its flexibility—calls to enforcement policy functions may also occur on the server. For example, in SEWIKI, a security-oriented blog/wiki that we have built using SE-LINKS, server-side mechanisms make policy enforcement both more convenient and efficient. SEWIKI represents documents using a tree-shaped data structure. Enforcing an access control policy on these documents often requires tree traversals from parent to child and child to parent. We use the higher-level abstractions provided in the server tier to reliably implement optimized access control checking on these trees, reducing the number of traversals necessary to enforce the desired security semantics. Implementing a similar optimization in the database would be both tedious and potentially inefficient. (More details on this in Section 4.) When labels describe policies other than access control, such as when tracking provenance [5] or information flow [30], server-side enforcement may be a requirement, because labels must be updated or checked during normal computations.

SE-LINKS was designed so that security-checking code can be modular and reusable—experts build libraries of security policies and label formats, and these can be used by different applications. As a case in point, we use the same access control library for both SEWIKI and SEPIKE. SE-LINKS’s approach is similar in spirit to microkernel-style library operating systems where common services are linked as libraries rather than implemented in separate components. Indeed, different applications can access the same backend database by using the same security libraries—database-side enforcement in both applications will invoke the same UDFs.

As a final benefit, SE-LINKS is relatively portable, since policy enforcement relies only on user-defined functions, a feature found in essentially all DBMSs. On the other hand, specialized facilities available in different DBMSs can be used by the compiler to improve performance without affecting the application. For example, we exploit Postgres’s support for rich user-defined types and found that they can provide improved performance.

In summary, the main contribution of this paper is a demonstration of how the uniform data model present in languages like LINQ, LINKS, and others can be extended with security policy enforcement that is reliable, modular, portable, and efficient. As evidence of these claims, we present an implementation of our approach in the LINKS programming language, which we call SE-LINKS, and demonstrate, through application experience, that SE-LINKS is expressive and, through experimental measurements, that it achieves good performance. SE-LINKS is freely available.

2. OVERVIEW

We begin by describing LINKS [10], the programming language on which SE-LINKS is based. Next we describe our two main contributions: the integration of FABLE into LINKS to ensure that security policies are reliably enforced, and compiling policy enforcement functions to database-resident user-defined functions for improving performance.

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 [13], or from an HTML form) is not always in a form most suitable for processing at the server, and likewise server-side objects must be mapped to and from relations or other database-side representations. These factors constitute an 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 LINQ [18] or Kleisi [35]. 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 Assured Application-level Enforcement

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 SESPine, our model medical-record management system. SESPine allows patients, health-care professionals, insurance providers, and others to create, edit and view records related to a patient’s case. There are two key security goals in SESPine: (1) confidentiality: a record may contain sensitive information that should only be viewed by authorized principals; (2) integrity: records should be modified only by authorized personnel, and all modifications should be properly logged for later audit.

The first novel feature of SELINKS is its use of the FABLE type system to ensure that security policies are correctly enforced. In FABLE, and consequently in SELINKS, implementing a security policy proceeds in three steps. First, we define the form of security labels which are used to denote policies for the application’s security-sensitive objects. Second, we define the enforcement policy functions that implement the enforcement semantics for these labels. Finally, we construct the application so that security-sensitive operations are prefaced with calls to the enforcement policy code. The key feature of FABLE is that by setting up our security policy in this way, type-correctness will guarantee complete mediation: no sensitive data can be accessed or manipulated without first consulting the appropriate enforcement policy function.

We elaborate on these three steps in the context of SESPine.

Security labels. SESPine security labels specify a group-based access control policy, with separate access restrictions for readers and writers of a record. Such labels are defined by the type Acl, an algebraic datatype (a.k.a. variant type) typical of functional languages (e.g., ML and Haskell):

\[\text{typeof} \text{ Group } = \text{Principal(} \text{String}\mid \text{Insurance} \mid \text{Admin} \mid \ldots\text{)}\]

\[\text{typeof} \text{ Acl } = (\text{read:} \text{List(}} \text{Group}, \text{write:} \text{List(}} \text{Group)}\]

Acl is a record type with two fields, read and write, that contain lists of groups authorized to read and modify a record, respectively. The Group variant type defines our various notions of group: Principal(x), stands for the group that contains a single user x; Insurance, is the group of users that work for an insurance company; and Admin, which include only the system administrators.

Given this label model, the (simplified) schema for our record database can be written as follows:

```haskell
var table handle = table “patientrecs” with
    (recid : Int,
     lab : Acl,
     text : String{lab}) from database “medDB”;
```

The table contains three columns. The first column is the primary key. The second column stores the row’s security label, having type Acl. The third column’s data has labeled type String{lab}, which states that it contains data (of type String) that is protected by the security label stored in the lab field of the table. This is a kind of dependent type [3], 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. For SESPine, we implement an authorization check in the following policy function:

```haskell
fun access(α)(cred, acl, x:α(acl)) policy {  
  if (member cred acl.read) {Just(unlabel(x))}  
  else {Nothing}  
}
```

The argument cred is the user’s login credential, having type Group; acl has type Acl; and x is data protected by the acl label, having type α(acl). Links type inference infers the types of the first two arguments, and the return type, which is Maybe α. The Links type Maybe t is a variant type consisting either of the value Nothing, or the value Just(x) where x has type t. The access function is polymorphic (a.k.a. generic) in the type of x—the type α can be instantiated with any type t, and thus access can be called with any labeled type. For example, we could pass in a String{acl} for x, in which case access would return a Maybe String; if we passed in an Int{acl}, it would return a Maybe Int, etc. Note that access 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 acl’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{acl} to String, making it accessible once returned from the function. Note that Links is a mostly functional language. The only mutable state in an application is in the DBMS. Thus, by coercing the type of x to String, access does not inadvertently grant write access to untrusted application code.

It is critically important that the unlabel operator is only in enforcement policy code. This is because the type system ensures that data with a labeled type like String{acl} is opaque to the main program. While a String can be printed, searched, etc., a String{acl} cannot; its values can never be inspected directly by application code. Therefore, to access the contents of labeled 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.

Thus the type system ensures that the only code that must be trusted with respect to security enforcement is the enforcement policy, and this code is relatively small—on the order of a 100–200 lines in our experience, compared to thousands of lines for the rest of the application. With relatively simple enforcement policy functions one can express a variety of security policies, including access control, information flow, provenance, and (with some extensions) automaton-based policies. These implementations are simple enough to admit formal proof of their correctness [33].

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 records in the database.

1 fun getSearchResults(cred, keyword) server {
  for (var row <- table.handle)
    where (var txtOpt = access_str(cred, row.lab, row.rowtext);
        switch(txtOpt) {
          case Just(data) → data ~ /.*\{keyword\}.*/
          case Nothing → false
        })
    [row]
}

The getSearchResults function runs at the server (as indicated 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 patientrecs table. In particular, the comprehension evaluates to a list (syntax [row]) including each row in the table (for(var row ← table.handle)) for which the where clause is not permitted to examine the contents of row.text directly because it has a labeled type String{row.lab}. 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 not granted access the labeled text field of the row, then access_str reveals the data and returns it (having Maybe String type). Lines 4–7 check the form of txtOpt. If the user has been granted access (the first case), then we check 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 Efficient, 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 patientrecs) so that each can be passed to a call to the local function. This is one of the main drawbacks of server-side enforcement of policies, which is typical of frameworks that use a uniform database-server model: enforcing a custom policy may require moving excessive amounts of data to the server to perform the security check. In the case of LINQ, queries that include calls to C# methods (like in our getSearchResults example), simply throw exceptions when they are evaluated, since these method calls cannot be translated to SQL.

The second contribution of SELINKS is to avoid this problem by compiling enforcement policy functions that appear in queries (like access) to user-defined functions (UDFs) that reside in the database. Queries running at the database can call UDFs during query processing, thus avoiding the need to bring all the data to the server. Most major DBMSs provide user-defined function languages, so while our implementation currently uses PostgreSQL, it should be adaptable to other settings.

We implement this approach with three extensions to the LINKS compiler. First, we extend it to support storing complex LINKS values (most notably, security labels like those of type Acl) 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 con-

1Our extensions are to revision r995 of LINKS.
struct a PostgreSQL user-defined type (UDT) for each complex SELINKS type possibly referenced or stored in a UDF or table [24]. Our UDT library allows arbitrary SELINKS values (aside from function closures) to be stored at the database. SELINKS automatically translates the server-side representation of values to PostgreSQL UDTs whenever these values pass into the DBMS tier.

At the top of the DBMS tier in Figure 1, we show the three columns that store SESEpine records. The lab column depicts storage of the Acl record, a structured SELINKS type. This value is compiled to a C struct that represents this label, described in further detail in Section 3.1.

### Compiling policy code to UDFs.

So that enforcement policy functions like access can be called during query processing, SELINKS compiles them to database-resident UDFs written in PL/pgSQL, a C-like procedural language (similar to UDF languages 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; we do not currently support higher-order functions (but have yet find a need for them to when running database-resident code). 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 3.3 describes the compilation process.

Figure 1 illustrates that UDFs are compiled from SELINKS 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; we give an example of this in Section 4.2.

### Compiling Links queries to SQL.

The final step is to extend the Links list comprehension compiler so that queries like the one in getSearchResults can call enforcement policy UDFs. 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 a string representation that will be converted to the native UDT representation automatically in the DBMS. Section 3.4 shows the precise form of the SQL queries produced by our compiler.

In summary, cross-tier label based security enforcement in SELINKS provides two main benefits. Through the use of Fable type-checking, we can ensure that all security-sensitive operations are meditated by the appropriate calls to policy enforcement code. By providing a uniform set of abstractions to enforce security policies, SELINKS allows programmers to easily implement enforcement mechanisms well-matched to the needs of their applications, while our novel cross-tier compilation strategy takes care of ensuring that enforcement code runs efficiently within the DBMS whenever this is needed.

### 3. IMPLEMENTATION OF SELINKS

In this section, we present the details of our implementation of cross-tier policy enforcement; details of the Fable extensions to the Links type system are discussed elsewhere [33]. We begin by presenting our approach to storing SELINKS values in the DBMS, used most notably to encode security labels. Then, we show how we compile SELINKS enforcement policy functions to user-defined functions. Finally, we present the compilation of SELINKS queries to SQL queries that can refer to compiled UDFs.

#### 3.1 Representing SELINKS values with UDTs

The standard Links implementation only permits scalar values to be stored in the database. In SELINKS, we use PostgreSQL’s support for user-defined types (UDTs) to implement a direct in-memory representation of structured SELINKS values in the database. This greatly improves efficiency when compared to the simpler (and more portable) approach of simply serializing SELINKS values to the DBMS as strings. The flexibility of PostgreSQL UDTs also allows us to mimic the server representation of values within the database and makes it easy to compile SELINKS functions to UDFs.

On the other hand, UDTs are essentially viewed as “blobs” by the DBMS and are thus opaque to query optimizers. We use manually-expressed relational mappings to deal with this problem in some cases, and also experimented with a relational mapping, inspired by work on mapping XML documents to RDBMSs [12]. We defer further discussion on these points to Section 3.5.

#### 3.2 UDT encoding

UDTs in PostgreSQL are created by writing a shared library in C and dynamically linking it with the database. PostgreSQL requires this library to implement three features: an in-memory representation of the type, conversion routines to and from a textual representation (allowing the type to be used in standard SQL queries), and functions for examining UDT values. Our SELINKS UDT library also exports an API with various low-level operations for constructing and accessing the encoded values; the UDFs produced by our code generator use these operations to manipulate the encoded values.

The in-memory representation for SELINKS values is based on a UDT called Value. This type is a tagged union that en-
capsulates one of several flavors of SELINKS structured type as well as base SQL types like int and text. Rather than describe the low-level C structures that implement each of these types, we focus on the higher-level API that our library implements. A fairly complete fragment of our API is shown in Figure 2. It shows the function prototypes for constructing and destructing the two main kinds of SELINKS structured values—variants (type Variant) and records (type Record)—along with specialized functions for lists (type List).

To illustrate, consider variant types. SELINKS values of variant type are represented as a Variant UDT, and are constructed using the function variant_init. For example, the value Principal('Alice'), a member of the Group variant type defined in Section 2.2, is constructed via the function call: variant_init('Principal',string_as_value('Alice')). Here, the first argument is the variant constructor represented as a string; the second argument is a Value that stands for the argument of the constructor. In this case, we pass in the string argument ‘Alice’ as the second argument, coercing it to a value using string_as_value. We also define parsers for each of our UDTs so that they can be constructed from string literals. PostgreSQL uses these parsers to implicitly convert textual representations of values to the appropriate types. For example, instead of calling variant_init we could call the appropriate parser with the string ‘Principal('Alice')’ to construct the value.

In addition to constructors, the API includes destructors for accessing the contents of each UDT. For variants, the function variant_arg projects out the argument of a variant. Pattern matching is compiled using the function variant_matches which tests if a variant matches a pattern specified as another variant value. For example, the following function call below returns true:

```
SELECT variant_matches('Principal('Alice')'),'Principal()')
```

(This example also illustrates how PostgreSQL implicitly invokes our parsers to coerce textual representations of SELINKS values into their in-memory representations as UDTs.) Figure 2 also shows the API for manipulating lists and records. The list_nil and list_cons functions are the usual constructors, while list_hd and list_tl decompose a list. The record constructors record_init1, record_init2, etc., take a specified number of field name/Value pairs, while record_proj projects out the named field.

Finally, we include functions like variant_as_value that promote values of specific types to the generic Value type. Usually, we also allow Values to be downcast to more specific types, after a runtime tag check.

### 3.3 Compilation of SELINKS code to UDFs

We have implemented a new code generator for SELINKS that compiles SELINKS functions to PL/pgSQL code, the most widely used of PostgreSQL’s various UDF languages. PL/pgSQL has a C-like syntax and is fairly close to Oracle’s PL/SQL. It would be straightforward to write code generators for other UDF languages, to support additional DBMSs (e.g., T-SQL for SQL-Server).

Code generation follows standard common techniques, which we illustrate by example. Figure 3 shows the (slightly simplified) code generated for the access enforcement policy function given in Section 2.2. A function definition in PL/pgSQL begins with a declaration of the function’s name and the names and types of its arguments. Thus, line 1 of Figure 3 defines a UDF called access that takes three arguments. The first argument cred is a textual representation of a user’s credential and has the built-in text type. The second argument, doclab, is a Record type that represents the Acl type (see Section 2.2). The final argument x stands for protected data of any type. The anyelement type allows us to translate usage of polymorphic types in SELINKS to PL/pgSQL, rather than requiring access functions specialized to a particular type. At line 2, we define the return type of access to be a Variant, since in this case we return a Maybe type.

In the body of the function, lines 4–8, we check if the user’s credential cred is mentioned in the acl.read field. We project from the acl using the record_proj at line 4, and then test list membership. The member function is itself a UDF compiled from SELINKS. We omit its definition. If this authorization check succeeds, at line 5 we return a value corresponding to the SELINKS value Just(x) by using the variant_init constructor. Notice that the unlabel operator that appears in SELINKS is erased from the compiled code — it has no run-time significance. If the check fails, at line 7 we return the nullary variant construct Nothing.

### 3.4 Invoking UDFs in Queries

The last step is to compile SELINKS comprehension queries to SQL queries that can include calls to the appropriate policy UDFs. Prior to our extensions, the LINKS compiler was only capable of handling relatively simple queries. For in-
stance, queries like our keyword search with function calls and case-analysis constructs were not supported. Our added support draws on work from SLinks [11] and Kleisli [35] for compiling comprehensions to SQL.

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. Consider the sub-query on lines 2–6. 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, a complex SLinks value; and the protected text S.text, respectively. The result of the authorization check is named tmp1 in the sub-query.

The compilation of the where-clause in the main query appears on lines 8–14. Recall from the SLinks query comprehension (shown in Section 2.2) that the where clause needs to first test if the authorization check revealed the contents of the protected string; if so it then checks whether the revealed string contains the keyword. Since the authorization check returns a variant type, we have to first destruct it using the pattern matching operation variant_matches provided by our UDT API. If this pattern match succeeds, we project out the contents x using the function variant_arg, followed by a downcast, to see if it contains the keyword using SQL’s LIKE operator. If either condition fails, the where-clause is false. Line 1 of the query selects and returns the relevant columns (i.e., excluding the T.tmp1 field).

3.5 Relational mappings for SELinks values

PostgreSQL UDT support is both easy to use and very flexible. However, UDTs are opaque to the query planners and can foil optimizations. To address this problem, we also implemented an automatic relational mapping for SELinks values, based on standard techniques for mapping inheritance hierarchies [1] and tree-structured (e.g., XML) data to relations [12]. In essence we “flatten” the components of a structured types (records and variants) into multiple rows in a table and express relationships between these components using foreign key constraints, e.g., by having children of a node “point” to their parent. This mapping can be used in lieu of the UDT-based mapping by via a compiler flag.

In addition to improving the portability of SELinks to DBMSs that do not support UDTs, the relational mapping has the advantage of exposing the structure of a complex SELinks value in a manner that can be exploited by query optimizers and table indices. However, the downside of relying exclusively on this approach is that when compiled UDFs dynamically allocate temporary data, this data must be stored as rows in temporary tables. Unfortunately, updates to tables during query processing can inhibit optimizations and managing temporary data is cumbersome. In many DBMSs, table handles are not first-class values, preventing parameterization of UDFs based on a temporary table ID. In the worst case, properly managing temporary table data reduces to implementing some kind of garbage collection.

While it may be possible to solve these problems, we have found that using a combination of encoding strategies works well. For example, we manually implement mappings for some application objects that require direct access in the database (like the document representations in SESPINE and SEWIKI, discussed in greater detail Section 4), and use the UDT-based mapping for security labels. For objects that do not require allocating temporary data in UDFs, or are never accessed by UDFs, the automatic relational mapping could be applied, though we leave it to future work to allow both the UDT and relational encodings to be applied to different data in the same application.

4. APPLICATION EXPERIENCE

We have developed two sizable applications with SELINKS, SEWIKI and SESPINE. SEWIKI is a blog/wiki inspired by Intellipsedia [28] and SKWEB [4], applications designed to promote secure sharing of sensitive information among U.S. intelligence agencies and the Dept. of Defense, respectively. SESPINE is an application inspired by the Spine [21], an online medical health record management system used by the National Health Service in the United Kingdom. The basics of SESPINE were described in Section 2.2, so we devote most of this section to discussing SEWIKI, with a few additional details about SESPINE at the end. Demos of both applications can be found on the WWW.

4.1 SEWiki Overview

SEWIKI enforces fine-grained confidentiality and integrity policies on structured, tree-shaped documents. SEWIKI’s approach is fairly general: its policies and enforcement strategy should be applicable to a wide variety of information systems, such as on-line medical information systems, e-voting applications, and on-line stores.

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.

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.

Our implementation SEWIKI consists of approximately 3500 lines of SELINKS code. It enforces a combined group-based access control and provenance tracking policy. Policies are expressed as security labels having type DocLabel, a record type shown below with two fields, acl and prov which represent the access control and provenance policies, respectively:

\[ \text{typename DocLabel = (acl: Acl, prov: Prov)} \]

(The Acl type definition was shown in Section 2.2; the definition of Prov is shown in Section 4.3.) We now discuss each aspect of SEWIKI’s policy in more detail.

4.2 Access control on structured documents

SEWIKI documents are defined as trees, where each node represents a security-relevant section of a document at an
arbitrary granularity—a paragraph, a sentence, or even a word. A local security label is associated with each node in the tree, with a hierarchical interpretation: the access control list included in a given node acts as a bound on the access of the node’s descendants. This model is consistent with typical multi-level security (MLS) document markup. For example, if one section $S$ of a document is originally marked as accessible to the Secret group, but later the entire document is deemed to be TopSecret (a group containing strictly fewer principals than Secret), then $S$ now effectively has Top Secret labeling. On the other hand, if the marking of the entire document were later downgraded to Unclassified, only those subsections of the document explicitly marked as Unclassified would be immediately visible; portions labeled Secret would still require explicit declassification for public access.

**Server-side representation.** When manipulating documents at the server, documents are represented using the datatype shown below:

```plaintext
typename Doc = {llab: DocLabel, lab: DocLabel, text: String[lab], children: List(Doc)[lab]}
```

The type Doc is a SELINKS record that represents documents as $n$-ary trees. The text field contains the text associated with the current node (possibly empty), and the children field contains the (possibly empty) list of the node’s children, themselves Doc records. (Note that there is additional information in Doc record not shown to assist with formatting.) The more interesting fields are llab and lab. The former is the local label for the current node, which defines the bound on the access control lists of the node’s children. To make our implementation efficient, we additionally store an exact label in field lab whose ACL contains the intersection of the ACL in llab with the ACLs in the llab fields of the node’s ancestors. Thus, to determine whether access to a given node is to be granted, the code needs only to consult the exact label, and make no further reference to other labels of other nodes. This is why the type of text is String[lab] and not String[llab]. Similarly, the children field is also protected by the label lab, to ensure that unauthorized users cannot traverse the structure of the document tree.

On the other hand, if the user wishes to change the local label on a node, the exact labels of the node’s descendants must be updated accordingly. Updating the label is only permitted for principals having write access to the node itself, as determined by the node’s exact label. In a document tree of type Doc{}, every node in the tree is protected by a label—the root node is protected by 1 and every node’s children are protected by labels too. If the contents of one of the nodes is to be modified by changing its local label, the appropriate enforcement policy function must first be called to grant access to that node. That policy function also takes care of recursively updating the exact labels of the node’s children.

**Database-side representation.** We use a custom encoding of documents when storing them in the database, which exposes some of their structure to make queries more efficient. The basic schema of the table that we use to store documents is shown below.

```plaintext
var doc_table = table "documents" with
(docid: Int, lab: DocLabel, text: String[lab],
llab: DocLabel, parent: Int, sibling: Int)
) from database "docDB";
```

The primary key for the doc_table rows is docid. The lab, text, and llab fields serve the same purpose as the fields having the same name and type in the Doc record type. The parent and sibling fields serve to encode Doc’s children field: parent is a foreign key to the docid of the node’s parent and sibling is a foreign key to the node’s sibling in the parent’s conceptual list of children. This encoding is typical for hierarchical data [1].

**Policy updates.** When a user navigates to a particular page, we retrieve the entire page from the docDB database and convert it to a Doc tree so long as the root node of the document is accessible to the requesting principal. This happens via a series of queries that acquire the document breadth-first, selecting all children of the nodes loaded in the previous iteration. The code that performs this retrieval is trusted to reconstitute the document properly (just as the compiler is trusted when the mapping is performed automatically). That said, this trusted code does need the extra privilege to use the unlabel operator, and the Doc datatype ensures that the resulting tree contains exactly the labelings that respect the hierarchical relationship between nodes.

Rather than load the entire document into memory, one optimization would be to perform security checking on the database, and only load those portions of the document visible to the current principal. In particular, when we submit a query to retrieve all children of a node with ID $n$, we also include a call to the access control function that filters out children to whom the requesting principal does not have access (according to their exact labels). If significant portions of a document are inaccessible, this approach can reduce load-times and reduce network traffic.

The drawback of this approach is that changes made to the in-memory document will have to be synchronized with the database version, which can be cumbersome. For example, if the user were to update the local label of a node, the exact labels of all children of the node must be updated, even those children to which the user does not have access. By keeping the entire document in memory, this update is straightforward to perform. Otherwise, the update must be synchronized when the document is written to the database, which we found tricky to implement. Moreover, our interactions with the Intellipedia [28] authors suggest that typically most of an MLS document will have a uniform marking, so the performance advantage of loading only part of a document may not be significant.

A key conclusion to be drawn from this discussion is that SELINKS type system and compilation strategy ensure that security policies are uniformly and reliably enforced, wherever that enforcement may take place. This allows the programmer to choose the strategy that best meets an application’s needs without worry about its impact on security enforcement.

**4.3 Data provenance tracking**

SEWiki maintains a precise revision history of a document in the labels of each document node—this is a form of data provenance tracking [5] that can be used to establish a document’s integrity (i.e., level of trust). This part of labels, having type Prov, is defined as follows:

```plaintext
typename Op = Create | Edit | Del | Restore | Copy | Relab
```
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 [33, 2].

Using security labels to represent provenance information provides two important benefits. First, since labeled data cannot be manipulated directly, we can ensure that all provenance-relevant operations are intercepted by enforcement policy code. This code can then perform the requested operation and update the provenance metadata on the results as necessary. Second, by expressing the relationship between data and its provenance in the types, we ensure that application code does not either confuse itself, or, worse, confuse the enforcement policy, by mistakenly associating the provenance of one datum with another.

### 4.4 Access control and provenance in SESpine

Our second application is a medical record management application, allowing physicians, patients, specialists, and administrators to create, modify, and share medical records. It consists of over 1000 lines of code specific to this program, sharing about 300 lines of policy code with SEWIKI.

When a user logs into the system, he or she must select a role from a list of appropriate roles. For example, a doctor who is both a physician and a patient must select one role or the other to begin with. We use an access control policy (shared with SEWIKI) to enforce role boundaries; administrators add patients to the system, and view billing fields, etc., but cannot view patients' medical data; physicians can create and modify patients' medical data, while patients are allowed to see their own medical data, but not to edit it.

A secondary privacy policy controls individual users’ access to fields. For example, a patient may not wish to allow all physicians to read his or her medical record, so the patient can modify his/her privacy policy to explicitly permit or deny access to certain physicians.

We implement a provenance policy (again shared with the SEWIKI) used to log all modifications to records within the system. This can be used to track which doctors made certain changes to a record, or to show by whom and when an address was modified.

We currently investigating other policies, inspired by healthcare policy scenarios from [22]; these include a signature policy where a medical record created by a physician can only be view by other physicians once the author has signed off on its validity. We are also investigating a meta-policy where, in an emergency situation, a physician can be granted access to an otherwise restricted medical record, with the caveat that extra logging is performed (in the form of additional provenance information).

## 5. EXPERIMENTAL RESULTS

We conducted a simple experiment to understand the performance benefits of compiling application-level policies to UDFs that run at the database, rather than the server. We also examine how much of an impact the location of the database (local host or networked) makes on server-side versus database-side enforcement. Our results show that executing SELINKS policy enforcement code at the database greatly reduces the total running time compared to running the same code at the server (almost a 10× speed increase).

### 5.1 Configuration

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

For our test, we used the `getSearchResults` query presented in Figure 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 a table of random 100,000 records, each comprised of 5–200 words selected from a standard corpus. Each record has a 5% probability of containing our keyword, and each record is labeled by a random access control label, which grants access approximately 10% of the time. Thus, the query returns approximately 500 of the 100,000 records. We examined two different policy enforcement scenarios: running the policy enforcement code on the server (by disabling UDF compilation), and running the enforcement code as a compiled UDF on the database. As a baseline, we also provide a comparison against a configuration that ignored the security checks altogether. All running times are the mean of five runs.

### 5.2 Results

The results of our experiment are summarized in Figures 5 and 6, which illustrate the time required to run the query using a local and remote networked database, respectively. The horizontal axis illustrates the policy enforcement location used (Server, Database, or None).

Both figures exhibit the significant improvement that comes from executing SELINKS policy code on the database rather than the server. For the local-database example we see a 5.5× improvement; for the the networked-database, the improvement is 9.5×.

Although we show a large speed increase of the database-side policy implementation over the server-side, it is important to note that the current incarnation of SELINKS is an
interpreted language with few optimizations; there are undoubtedly more ways to optimize the server portion directly. However, the speed difference between the server enforcement for local and networked databases shows a substantial overhead that is likely to be independent of any server-side improvements.

A comparison against the baseline shows that the cost of enforcing a security policy is also significant. This comparison is somewhat artificial, since trading security for performance is clearly not an option. However, we believe there is considerable room for applying further optimizations to our code generators and data layouts. For instance, as mentioned in section 3.5, better use of relational mappings may allow the DBMS to better optimize our queries. We have also begun exploring the use of more dense encodings of primitive SELinks types when they are used in the database.

In summary, running SELinks policies on the database instead of the server greatly improves performance, particularly for queries over a network. However, both the server and database components of SELinks have the potential to benefit from more optimization.

6. RELATED WORK

SELinks is an extension of Links [10], a programming language similar to LINQ [18], Hop [15], Volta [34], GWT [14], Ruby on Rails [26] and others in that it provides a uniform model for programming each tier of a multi-tier web application. SELinks extends this model with novel security mechanisms that enable efficient enforcement with high assurance. Some uniform model frameworks (such as Java EE [16]) also provide abstractions for expressing security policies on application objects, though with rather different mechanisms. While these abstractions are flexible and relatively easy to use (thanks in part to the lack of impedance mismatch), compared to SELinks they provide less assurance (since the type system provides fewer guarantees) and more overhead, since complex security-sensitive operations usually run at the server, requiring potentially large amounts of data to be transferred from the DBMS unnecessarily. SELinks addresses the first shortcoming through type-based verification, and the second by providing a novel cross-tier compilation mechanism that allows complex operations to be run entirely within the database. Our performance measurements corroborate the findings of Müller et al. [19] who report improvements in performance by orders of magnitude when application-level data management in consolidated within the DBMS, particularly when the DBMS and server are on separate networks.

An alternative to expressing and enforcing security policies at the level of application objects is to express policies in terms of database-level objects, using DBMS-provided facilities. Mechanisms for this purpose differ depending on the DBMS. For example, PostgreSQL [25], SQLServer [32], and MySQL [20] all provide security controls that apply at the level of tables, columns, or stored procedures. Oracle 10g [23] and IBM DB2 [6] provide native support for schemas in which each row includes a security label that protects access to that row, where labels are in the style of lattice-based multi-level security classifications. A common approach to row-level security in other DBMSs is to define a parameterized view [29] of a table that is based user-specific criteria [27]. This view essentially filters out the rows to which the current user is denied access.

The benefits of using DBMS-resident facilities are twofold: (1) application code does not need to be trusted to perform the security checks correctly since these are handled by the DBMS, and (2) DBMS-side checking can be more efficient than server-side checking. The main drawback is a lack of flexibility: DBMS mechanisms may not match the needs of an application. For example, 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 for an application like SEWiki. By contrast, security policies in SELinks are specified for application-level objects (which can be at the table, row, or even cell level, when viewed from the database’s perspective) using essentially arbitrary encodings of labels. Moreover, SELinks’ type system helps regain assurance of correct enforcement, while compilation of enforcement policy functions to UDFs can result in DBMS-side enforcement with its attendant performance benefits. Indeed, SELinks’ approach to compiling authorization code to UDFs that filter query results is quite similar in spirit to using views, but in a manner that is easier to use, since there is no impedance mismatch, and more reliable, since the type checker will ensure enforcement policy functions are called when necessary.

Type-based assurance of correct security policy enforcement in SELinks resembles the checking provided by languages like Jif [8] and FlowCaml [31]. However, neither
of these languages provides integrated support with a database, creating an impedance mismatch, and both languages focus exclusively on enforcing information flow policies, whereas SELINKS can support the enforcement of these and other styles of policy. Swift [7] and SIF [9] are two frameworks that have been built using Jif to address various aspects of security when constructing multi-tier web applications. However, both languages essentially ignore the database tier (SIF focuses on servlet interactions and Swift considers client-server interactions).

Rizvi et al. [29] also address the problem of checking that queries contain the appropriate authorization check. Their approach requires an administrator to specify a security view to filter the contents of a table. Application code can then issue arbitrary queries on the behalf of users against the unfiltered table. Their system runs the query only after checking that the query can be run against the filtered view of the table. Unlike our approach, their security enforcement mechanism is transparent—unauthorized users are unaware that their queries are actually being run against a filtered table. However, transparency is not always desirable. When particularly complex policies are in effect, it is often important to explain why authorization checks failed both for diagnosis and so that users can attempt to revise their requests with the appropriate credentials to gain access. Furthermore, rather than refusing to run queries at runtime, queries in SELINKS are checked statically to ensure that they contain the appropriate checks, promoting early detection of programming errors.

7. CONCLUSIONS

This paper has presented SELINKS, a programming language focused on building secure multi-tier web applications. SELINKS provides a unified view of security enforcement for programs that span the server-database divide. Through the use of a novel type system, SELINKS ensures that security policies are correctly enforced. Our type system statically detects errors such as missing authorization checks, whether these errors occur in server or data-access code. In order to support this unified model of security enforcement while still retaining good performance, SELINKS compiles policy enforcement code to database-resident user-defined functions, which can be called directly during query processing. This can minimize the overhead of data transfer between the server and database. Our experience with two sizeable web applications, a model health-care database and a secure wiki with fine-grained security policies, indicates that SELINKS is flexible, relatively easy to use, and, when compared to a single-tier approach, improves throughput by nearly an order of magnitude.

8. REFERENCES


