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

Brian J. Corcoran[†] Nikhil Swamy[‡] Michael Hicks[†]

[†]University of Maryland
College Park, MD, USA
{bjc,mwh}@cs.umd.edu

[‡]Microsoft Research
Redmond, WA, USA
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 in the database or at the server. Object-level security policies are expressed as fully-customizable, first-class *labels* which may themselves be subject to security policies. 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 implements a type system called Fable that allows a protected object's type to refer to its protecting label. The type system can check that labeled data is never accessed directly by the program without first consulting the appropriate policy enforcement function. Second, SELinks compiles policy enforcement code to database-resident user-defined functions that 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 sizable 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. SELinks is freely available.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; C.2.4 [Distributed Systems]: Client/server; H.2.0 [Database Management]: General

General Terms Security, Languages, Performance

Keywords database programming, security enforcement, web applications, type systems, compilers

Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This work was also supported in part by NSF grant CCF-0524036.

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 is 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 language 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 [38], the Google Web Toolkit [16], and Hop [17] (unifying the client-server view), and LINQ [20, 19], Java EE (formerly, J2EE) [18], and Ruby on Rails [30] (unifying the server-database view).

Our focus in this paper is to explore how to reliably build multi-tier web applications that handle private or sensitive data. The uniform server-database programming model provided by web application frameworks 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. For example, when processing a query on a table that contains sensitive data, the server-side enforcement strategy requires *all* potentially-relevant results to be transferred from the database to the server so that the server 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 seldom-taken path. If several applications must access the same back-end database, these security checks must be repeated in each application, which increases both the programming burden and the chances for mistakes.

This paper presents *Security-enhanced* Links (or SELinks), an extended version of the Links multi-tier programming language with novel support for fine-grained security policy enforcement. SELinks 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), SELinks 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, SELinks implements a novel type system called Fable [37]. The Fable declaration o : *Int*{l} ascribes object o a *labeled type* *Int*{l}, which indicates in this case that o is an integer protected by 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*{l} 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(l:*Lab*, data:*Int*{l}, 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 granted access to it, according to l. By making the types of enforcement policy functions suitably specific, we can essentially ensure *complete mediation*: a labeled datum *must* be passed through a particular policy function that performs the appropriate policy check before that datum can ever be used.

To ensure good performance, when calls to enforcement functions occur in database queries, the SELinks compiler translates the calls to user-defined functions (UDFs) accessible during query processing. For example, we have implemented a simple web-based application called SESpine, inspired by *The Spine* [24], 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.

Our 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 SELinks, 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.) Additionally, SELinks gives programmers the flexibility to choose security labels that match the needs of their applications and apply them to objects at an arbitrary granularity. For example, in addition to enforcing access controls, we use labels in SEWiki to track fine-grained provenance information [5] and rely on server-side enforcement to update labels to account for data flows through normal computations that occur at the server.

SELinks 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 SESpine. SELinks' approach is similar in spirit to microkernel-style *library operating systems* [14] where common services are linked as libraries rather than implemented in separate components. Indeed, different applications can access the same back-end database by using the same security libraries—database-side enforcement in both applications will invoke the same UDFs.

As a final benefit, the general approach of policy enforcement in SELinks is relatively portable since our cross-tier compilation strategy relies only on user-defined functions, a feature found in essentially all major 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 PostgreSQL's support for rich user-defined types and found that they can provide improved performance. For DBMSs that do not provide such a facility, simpler solutions (with some degradation in performance) are possible.

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 SELinks, and demonstrate, through application experience, that SELinks is expressive and, through experimental measurements, that it achieves good performance. SELinks is freely available at `http://www.cs.umd.edu/projects/PL/selinks`.

2. OVERVIEW

We begin by describing Links [10], the programming language on which SELinks 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. Finally, we discuss how through the use of SELinks' type system programmers can flexibly enforce custom security policies at an arbitrary granularity while using our cross-tier code generation tools to optimize the performance of their applications.

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 [15], 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 [20] or Kleisli [39]. 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 Reliable enforcement of security policies

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, and insurance providers 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.

In order to allow an engineer to verify that a security policy is correctly implemented, SELINKS implements a type system called FABLE. 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. A security label l is associated with the object o it protects by giving o a type that mentioned l, e.g., $t\{l\}$. Second, we define the *enforcement policy* functions that implement the security semantics of these labels. Finally, we construct the application so that security-sensitive operations on objects of type tl are always prefaced with calls to the enforcement policy code. A key property of FABLE is that type-correctness guarantees complete mediation: no sensitive data can be accessed or manipulated without first consulting the appropriate enforcement policy function. We elaborate on the three steps involved in implementing a FABLE-style security policy 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):

```
typename Group = Principal(String) | Insurance | Admin
typename Acl = (read:List(Group), write:List(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. Among others, the *Group* type includes the singleton group Principal(x) containing only a single user x; Insurance, the group of users that work for an insurance company; and, Admin, the group that includes only the system administrators.

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

```
var table_handle = table "patientrecs" with
    (recid : Int,
     lab : Acl,
     text : String{lab}) from database "medDB";
```

The table above 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] which, 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:

```
sig access: (cred:Cred, lab:Acl, data:α{lab}) → Maybe(α)
fun access(cred, lab, data) policy {
  if (member(cred, lab.read)) { Just(unlabel(data)) }
  else { Nothing }
}
```

A function declaration in SELINKS can optionally be prefaced by a **sig** declaration, which specifies a type signature for the function. The **sig** declaration above shows access to be a function of three arguments. The first argument cred is a user credential of type *Cred*; the second argument lab is a security label of type *Acl*; the final argument data is given the type α\{lab\}, which indicates that it is protected by the label lab (the second argument), and can only be accessed after the suitable access control check has been performed. The signature also indicates that the value returned by access is of type *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 type of data and the return type indicate that the access function is *polymorphic* in the type of data—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 value of type *String*{lab} for data, in which case access would return a *Maybe*(*String*); if we passed in a *Int*{lab}, 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 lab'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*{lab} to *String*, making it accessible once returned from the function. Note that in Links, by default, all data is immutable, so coercing the type of x to *String* in access is not granting write access. The only mutable state in an application is in the database, and update operations on tables are mediated separately.

The SELinks type system ensures that the **unlabel** operator (and a corresponding **relabel** operator) is only ever used in code that has been granted a special privilege by being marked with the **policy** keyword. This allows us to prove that the unprivileged application code always treats labeled types abstractly. For example, consider data with a labeled type like *String*{lab}. While a *String* can be printed, searched, etc., a *String*{lab} 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 (e.g., **unlabel**) or deny access to the data. If application code directly calls **unlabel** or attempts to directly access a labeled type, the program is rejected by the type checker at compile time. Note that the privileged **unlabel** and **relabel** operators are only used to type check programs and have no operational significance; they are erased during code generation.

The SELinks type system allows us to conveniently partition the code that must be trusted to perform security enforcement from the rest of the program. In practice, we have found the enforcement policy fragment of program to be relatively small—on the order of a 100–200 lines in our experience, compared to thousands of lines for the rest of the application. The enforcement policy can be subject to further manual inspection or even formal verification to ensure that it correctly implements the desired security semantics. We have shown that 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 formally verified proofs of correctness [37, 36].

Including policy checks in queries. The final step is to preface security-sensitive operations on data with calls to enforcement policy functions, whether these operations occur in server-side code, or in data access code. Here is a function that performs text search on the records in the database.

```
1. fun getSearchResults(cred, keyword) server {
2.    for (var row ← table_handle)
3.    where (var txtOpt = access_str(cred, row.lab, row.text);
4.              switch(txtOpt) {
5.              case Just(data) → data ~ /.*{keyword}.*/
6.              case Nothing → false
7.              })
8.    [row]
9. }
```

Note that not all SELinks functions need to be given explicit type signatures using **sig** declarations. Where the types in a function's interface do not contain any label dependences, the SELinks type checker gracefully degrades to use Links' underlying type inference algorithm. In this case, our type checker infers that the type of getSearchResults is (*Cred*, *String*) → *List*(*Row*), where *Row* is the type of a row in in the `patientrecs` table.

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 true. 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 authorized to 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 large 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[1]. 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.

Figure 1 illustrates these three elements. The figure depicts the server running the SELinks program on the left,

[1]Our extensions are to rev.995 of Links-0.4 from May 2007.

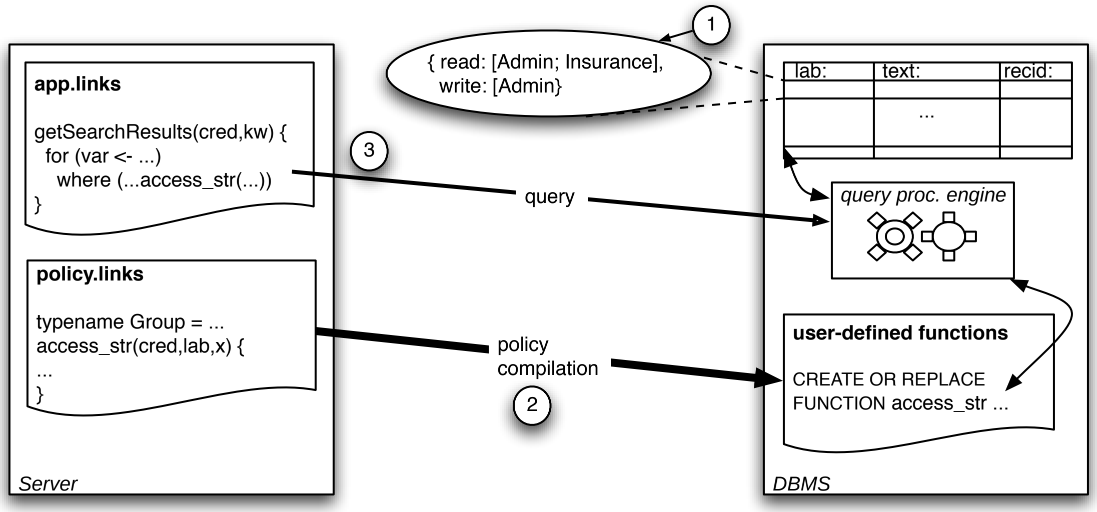


Figure 1: Cross-tier Policy Enforcement in SELinks

and the DBMS, processing our example query, on the right.

Storing complex SELinks **data in the DBMS.** The DBMS contains the `patientrecs` table that stores SESpine records. The lab column stores *Acl* values, which have complex SELinks type (labeled (1) in the figure). The other two columns store the record text and a document index. Our Links compiler extension uses common DBMS support for native values, in particular employing PostgreSQL support for *user-defined types* (UDT). We extended the Links compiler to construct a UDT for each complex Links type possibly referenced or stored in a UDF or table [28]. The engineer initially populates the table using the expected representation, and SELinks automatically translates the server-side representation of values to PostgreSQL UDTs whenever these values pass into the DBMS tier. We discuss the details of our UDT representation and other alternatives we considered (in particular, serialized strings and indexed records following an object-relational (OR) mapping) in Section 3.1.

Compiling policy code to UDFs. So that enforcement policy functions can be called during query processing, they are compiled to database-resident UDFs. For our example, enforcement policy functions like access_str are defined in the file policy.links, and as shown in the figure (labeled (2)), the developer directs these functions to be compiled to UDFs. SELinks extends the Links compiler with a code generator for PL/pgSQL, a C-like procedural language similar to UDF languages available for other DBMSs. The generated code uses the UDTs defined above 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 PostgreSQL UDTs implemented as C structs. Section 3.2 describes the compilation process.

Compiling Links **queries to refer to UDFs.** Once enforcement policy functions are resident at the DBMS, they can be invoked directly during query processing. The figure illustrates that the list comprehension in app.links can be compiled to SQL that directly invokes the compiled access_str function now resident at the DBMS, where any Links values of complex type are converted automatically when the UDF is invoked. Section 3.3 shows the precise form of the SQL queries produced by our compiler.

Note that the policy code in policy.links also resides on the server and the application app.links can, when enforcing policies on server objects, simply call the server version of the policy. We discuss an example that relies on this code replication feature in Section 4.2.

2.4 Discussion

By selecting an appropriate language of security labels and applying them to program objects at an arbitrary granularity, SELinks programmers can design security mechanisms that closely match the needs of their applications. The type system and compilation strategy ensure that policies are uniformly and reliably enforced, whether the enforcement occurs at the database or at the server. In this section, we briefly discuss the expressiveness of SELinks' security enforcement mechanisms and compare it to typical label-based row-level security protections provided by DBMSs like Oracle 10g [26] and IBM's DB2 [6].

Customizable label models. By allowing arbitrary data values to be used as security labels, SELinks policies can be highly specialized to an application. For example, SESpine's label model includes specific roles relevant to the medical setting, and maintains separate read and write privileges. SEWiki, our blog/wiki application discussed in Section 4, extends SESpine's label model to track provenance information. In contrast, a built-in labeled security mechanism such as Oracle Label Security (OLS) [26] only provides a single multi-level security model in which labels are integers arranged in a total order. DB2's row-level security mechanisms [6] offer a more flexible label model; however, unlike in SELinks, these labels must always be arranged in a lattice. In both these cases, the specialized label models, implemented natively in the DBMS, provide optimized support for applications that fit the paradigm of lattice-based multilevel security (e.g., US military applications). However, the native label models do not obviously meet the needs of other applications.

On the other hand, one advantage of SELinks' relatively high level of abstraction is that it can employ the native label models of existing DBMSs, if the programmer so chooses. For example, we have begun experimenting with interfacing with OLS from SELinks [2]. OLS' built-in label values can be represented as integers in SELinks and labeling relationships can be reflected on database objects using SELinks' type language. This approach can be used to augment the native DBMS protections offered by OLS with features like information flow tracking throughout a web application, e.g., through various server and client operations.

Policies of varying granularity. SELinks policies can be applied at an arbitrary granularity. For example, in SESpine, we choose to label each `text` field in each row of the `patientrecs` table with its own security label, i.e., we use row-level controls. In SEWiki, we use a hierarchical tree-shaped data model in which labels apply to entire sub-trees, corresponding to several rows in table. It would also be straightforward to apply a security label at a much coarser granularity, e.g., a label could be used to protect an entire table.

Although the application designer is free to choose a granularity to match her application, finer-grained policies are more expensive to enforce. For example, in the function getSearchResults of Section 2.2, a policy check must inspect the label of each row in the table. This behavior is not peculiar to SELinks—a native implementation of row-level security like OLS must likewise perform an access check on each row. For a coarser policy such as table-level controls, a single check of a table's label would suffice for each query.

The combination of a custom label model and policies applied at a programmer-chosen granularity, matched with the use of enforcement policy functions to mediate all security sensitive operations allows for a very broad range of policy styles to be expressed and enforced in SELinks. A detailed formal analysis of the expressiveness of SELinks' policy enforcement model can be found in Swamy's thesis [36].

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 [36]. We begin by presenting our approach to storing SELinks values in a 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

```
Variant variant_init(text, Value);
Value variant_arg(Variant);
bool variant_matches(Variant, Variant);

List list_nil();
List list_cons(Value, List);
Value list_hd(List);
List list_tl(List);

Record record_init1(text, Value);
Record record_init2(text, Value, text, Value);
Value record_proj(Record, text);

Value variant_as_value(Variant);
Value list_as_value(List);
Value record_as_value(Record);

Variant value_as_variant(Value);
List value_as_list(Value);
Record value_as_record(Value);

text value_as_string(Value);
Value string_as_value(text);
```

Figure 2: PgSQL/C API for SELinks UDTs (partial)

to UDFs. Another option would be to store SELinks values in the DBMS using an object-relational mapping; we discuss this option in Section 3.4.

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 in-memory representation for SELinks values is based on a UDT called `Value`. This type is a tagged union that represents one of several flavors of SELinks structured type and base SQL types like `int` and `text`. A fragment of the API exposed by our UDT library is shown in Figure 2. Our API contains functions 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`).

For an illustration, 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 SELinks value Principal("Alice"), an instance of the *Group* variant type defined in Section 2.2, is constructed via the C 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`. In general, our API includes functions like `string_as_value`, `variant_as_value`, etc. that promote values of specific types to the generic `Value` type. Dually, we also allow `Values`

```
 1. CREATE FUNCTION
 2.   access(cred text, lab Record, data anyelement)
 3. RETURNS Variant AS $$
 4. BEGIN
 5.   IF member(cred,record_proj(lab, "read")) THEN
 6.     RETURN variant_init('Just', data);
 7.   ELSE
 8.     RETURN variant_init('Nothing', null);
 9.   END IF;
10. END;
11. $$ LANGUAGE 'plpgsql'
```

Figure 3: Generated PL/pgSQL code for `access`

to be downcast to more specific types using the functions `value_as_string`, `value_as_variant`, etc.

The API includes destructors for extracting 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 SQL select statement includes a call to our pattern matching function; this query evaluates to the result `true`:

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

PostgreSQL can implicitly parse a textual representation of a value to its an in-memory UDT representation. To use this facility, we define parsers for each of our UDTs. For example, instead of calling `variant_init` explicitly, PostgreSQL uses our parsers to automatically translate the string value `"Principal('Alice')"` to the appropriate `Variant` structure.

Finally, 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.

3.2 Compiling SELinks programs 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 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 compilation 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, lines 1–2 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, `lab`, is a `Record` type that represents the `Acl` type (see Section 2.2). The final argument `data` stands for protected data of any type. The `anyelement` type allows us to translate usage of polymorphic types in SELinks to

```
SELECT recid, lab, text FROM
  (SELECT
     P.recid AS recid,
     P.lab AS lab,
     P.text AS text,
     access('Alice', P.lab, P.text) AS tmp1,
  FROM patientrecs AS P) AS T
WHERE
  CASE
    WHEN ((variant_matches(T.tmp1, 'Just((_))')))
      THEN (value_as_string(variant_arg(T.tmp1))
           LIKE '%keyword%')
    WHEN (true)
      THEN false
  END
```

Figure 4: SQL query generated for getSearchResults

PL/pgSQL, rather than requiring access functions specialized to a particular type. At line 3, 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 5–9, we check if the user's credential `cred` is mentioned in the `lab.read` field for the *Acl* type. We project from `lab` using the `record_proj` at line 5, 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 6 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—as described in Section 2.2, it has no run-time significance. If the check fails, at line 8 we return the nullary variant construct Nothing.

Our code generator currently can translate source programs that are in a fragment of SELinks that corresponds essentially to a first-order functional language, with recursion, variants and pattern matching, and records. Notably, we do not generate code for programs that use higher-order functions, query comprehensions, or Links' multi-threaded message passing constructs. In practice, we only compile enforcement policy functions to UDFs. Since these functions are part of trusted security infrastructure, they are, by design, inherently simple and, in the cases we considered, can be expressed in the limited fragment of SELinks that our compiler can handle.

Rather than compile SELinks code to PL/pgSQL, we could have developed a module that could run SELinks code directly at the DBMS as a PostgreSQL custom procedural language. Modules for running Python and Perl code at the DBMS have been developed previously. Our current approach has the advantages of performance and portability. In particular, our generated PL/pgSQL code is further optimized by the PostgreSQL optimizing compiler, and PL/pgSQL is similar enough to others DBMS UDF languages (Oracle's PL/SQL or SQL Server's T-SQL) that retargeting our code generator would be straightforward.

3.3 Invoking UDFs in queries

The final element of our cross-tier enforcement mechanism is to compile SELinks queries expressed as list comprehensions to SQL queries that include calls to the appropriate enforcement policy UDFs. 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. Our added support draws on the existing query compiler in Links compiler [13], which, in turn, draws on Kleisli [39], for compiling comprehensions to SQL.

Figure 4 shows the SQL generated by our compiler for the keyword search query in the body of getSearchResults function of Section 2.2. 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 2–7. Lines 3–6 select the relevant columns from the `patientrecs` table; line 7 calls the policy function `access`, passing in as arguments the user credential; the document label field `P.lab`, a complex SELINKS value that represents the *Acl* datatype; and the protected text `P.data`, 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 9–15. Recall from the SELINKS 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 checks whether the revealed string contains the keyword. Since the authorization check returns a variant type, we have to destruct it using the pattern matching operation `variant_matches` provided by our UDT API. If this pattern match succeeds, we project out the contents `data` 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).

3.4 Discussion: Query optimizations

PostgreSQL's UDT support is both easy to use, very flexible and provides good performance for UDFs because structured values can be manipulated using a compact in-memory representation, rather than using cumbersome textual representations. While UDTs are optimized for use in UDFs, when structured values need to be inspected directly in SQL queries, a UDT-based representation can be quite inefficient. Since the structure of a UDT value is, in general, opaque to the query planner, standard optimizations necessary for good performance are foiled. In this section, we discuss shortcomings of our current implementation and present directions for future work on optimizing the representation of structured values that may help alleviate these concerns.

Generating indices on UDT columns. For large tables, query performance depends crucially on well-chosen indices. For example, a common query on the `patientrecs` table in SESPINE is to retrieve all records to which a given principal, say Alice, has access. A simple implementation of this query in SELINKS might be of the form

```
for (var row ⟵ table_handle)
  where (member aliceCred row.lab.read)
  [row]
```

In our current implementation, a query of this form requires a full table scan since we represent row.lab using an opaque UDT that cannot easily be indexed.

PostgreSQL, however, does include a feature that can be used to generate limited forms of indices on UDTs. By providing an API that defines an equality predicate, an ordering relationship, and a hashing function for UDTs, PostgreSQL can be directed to index a table based on a column that stores a UDT. For our applications, this simple facility may be sufficient to improve performance substantially. Recall that we generally use UDTs to only store security labels in the database. These labels can often be arranged in lattices [12, 22] that can be used to form the basis of a partial order that underlies an index.

We are currently investigating the possibility of automatically generating indices for columns that store security labels. We envisage that the programmer will provide SELINKS implementations of functions that define the partial order on labels. For example, for the *Acl* type of Section 2.2, the programmer can implement functions that define the partial order based on a subset relationship of the user-lists stored in the read and write fields of a pair of *Acl* values. By compiling these functions and loading them into the DBMS, we hope to generate suitable indices for UDTs. With this facility, many common queries on structured data values (e.g., testing security label inclusion for lattice-based labels) can be performed without a full table scan.

Relational mapping for SELINKS values. Generating indices for UDTs can provide good performance when queries only use specialized operations that are consistent with the partial order used by the index. More generally, queries may contain arbitrary operations on structured SELINKS values, e.g., a query on the `patientrecs` table may search for all records in which the set of writers is greater than the set of readers. An index based on a partial order over *Acl* labels does not help with such a query.

To address this problem, we have implemented an automatic relational mapping [1] for SELINKS values so that a complex value is represented in multiple rows of a table using only native SQL types. 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 exposing the structure of a complex SELINKS value to query optimizers and table indices, a relational mapping has the advantage of improving the portability of SELINKS to DBMSs that do not support UDTs. 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 parametrization of UDFs based on a temporary table ID. In the worst case, properly managing temporary table data reduces to implementing a kind of garbage collection.

Undoubtedly, lifting these limitations requires considerable additional work. However, we have found that using a combination of the existing mechanisms in SELINKS works acceptably well. For example, we manually implement mappings for some application objects that require direct access in the database (like the representation of records in SESPINE and structured documents in SEWIKI, discussed in greater detail Section 4), and use the UDT-based mapping (without indices) 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 *Intellipedia* [32] and SKIWEB [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* [24], an on-line 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 SELinks web site.

4.1 Overview of SEWiki

SEWiki enforces fine-grained confidentiality and integrity policies on tree-structured documents. SEWiki's approach is fairly general: its policies and enforcement strategy should be applicable to a variety of information systems, such as online 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 document fragments, allowing 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 *provenance* information can be used to assess the trustworthiness of data in a document and can also be used to conduct an audit when information is leaked or degraded.

Our implementation of 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:

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

(The *Acl* type definition was shown in Section 2.2; the definition of *Prov* is shown below, 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 single 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 TopSecret labeling. On the other hand, if an entire document was marked Secret and later downgraded to Unclassified, portions of the document specifically labeled TopSecret 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:

```
typename Doc = (local_lab:DocLabel,
                exact_lab:DocLabel,
                text:String{exact_lab},
                children:List(Doc){exact_lab})
```

The type *Doc* is a SELinks record with four fields 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 local_lab and exact_lab. The former is a label associated with the current node in the document tree, 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 exact_lab. This field is a *DocLabel* in which the acl field contains intersection of the acl field of the nodes local_lab with the exact_lab field of the node's parent. Effectively, the exact_lab field at a given node memo-izes the local labels of all the nodes on the path from that node to the root of the document tree. 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.

To indicate that the text field contains sensitive data, we give it a labeled type *String*{exact_lab}. Note specifically that the type of text is not *String*{local_lab} since this would fail to ensure that a parent node's access restrictions were properly applied to its children. For the same reason, the children field is given the type *List*(*Doc*){exact_lab} rather than *List*(*Doc*){local_lab}. Furthermore, notice that giving children the type *List*(*Doc*{exact_lab}) is inadequate since this type indicates that the list of children can be traversed by code acting on the behalf on any principal. By giving the children field the type *List*(*Doc*){exact_lab}, we ensure that even the structure of the document tree (and not just its contents) is accessible only to authorized principals.

The use of exact_lab simplifies access control checks to a given tree node, but updates to local labels will impose a non-constant overhead. In particular, if a principal wishes to change the local label on a node, the exact labels of the node's descendants must be updated accordingly. Since policy updates are relatively rare compared to document accesses, our implementation is optimized towards providing fast access to documents while updates are relatively slower.

Only a principal with the appropriate write-privilege is permitted to update a node (whether it be the node's contents or the node's label). In a document tree of type $Doc\{\mathsf{l}\}$, every node in the tree is protected by a label—the root node is protected by l and every node's children are protected by labels too. If the contents of one of the nodes is to be modified, the appropriate enforcement policy function must first be called to grant access to that node. If the update causes a change to a node's local label, then the policy function takes care of recursively propagating the update to 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 and is a standard relational mapping for hierarchical data [1].

```
var doc_table = table "documents" with
  (docid : Int, exact_lab : DocLabel,
   text: String{exact_lab}, local_lab: DocLabel,
   parent: Int, sibling: Int
) from database "docDB";
```

The primary key for the doc_table rows is docid. The fields exact_lab, text, and local_lab 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

Mapping between server and DB representations. When a user navigates to a particular page, we retrieve the entire document 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 document nodes 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—it must, for example, ensure that a document node n that is represented as a row in the database with a foreign key relationship to a parent row p, is in fact placed in the children list of the node corresponding to p. In other words, the server code that fetches and persists a *Doc* value from/to the DBMS must perform a faithful object-relational mapping for tree data. We are currently exploring the use of SELINKS' type system to verify that this object-relational mapping is correctly performed.

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 submitting a query to retrieve all children of a node with ID n, we can 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.

While earlier versions of SEWIKI did implement this behavior, one drawback of doing so is that changes made to the in-memory document must be synchronized with the database version, which requires some additional bookkeeping. 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 more easily performed.

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:

```
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.

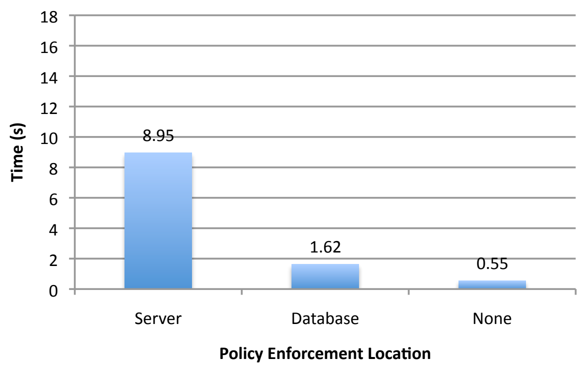
This provenance model exploits SELINKS' support for custom label formats. It is hard to conceive of encoding such a complex label format using native database support for row-level security. Finally, this policy does not directly attempt to protect the provenance data itself from insecure usage. We have shown elsewhere that protecting provenance data is an important concern and is achievable in SELINKS without too much difficulty [37, 11].

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.

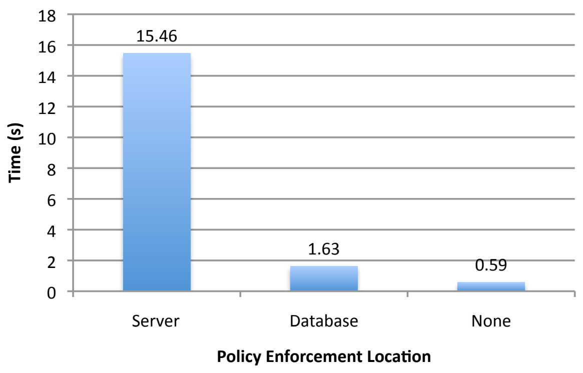
4.4 Access control and provenance in SESpine

SESPINE, originally introduced in Section 2.2, 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; ad-



(a) Local DB



(b) Remote DB

Figure 5: Keyword search, 100k rows

ministrators add patients to the system, view billing fields, etc., but cannot view patients' medical data; physicians can create and modify patients' medical data, while patients are allowed 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 real-world health-care policy scenarios [25]; 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. PERFORMANCE EXPERIMENTS

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 examined 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 100,000 randomly generated 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 represented in the DB using the UDT mappings of Section 3.1. These randomly chosen labels grant 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.

	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

5.2 Experimental results

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

Both graphs show that there is a significant performance benefit to 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×. A comparison against the baseline shows that the cost of enforcing a security policy is also significant (though the comparison is somewhat artificial, since trading security for performance is not a viable option).

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. Furthermore, as discussed in Section 3.4, many aspects of the SELinks' data layout and code generator can be optimized, to further improve the performance of programs that enforce their policies at the database.

6. RELATED WORK

SELinks is an extension of Links [10], a programming language similar to LINQ [20], Hop [17], Volta [38], GWT [16], Ruby on Rails [30] 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 [18]) 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 and more overhead. SELinks' type-based verification assures security checks are performed correctly, and its cross-tier compilation mechanism allows these checks to be run entirely within the database, avoiding the transfer of potentially large amounts of data from the DBMS to perform the checks at the server. Our performance measurements corroborate the findings of Müller et al. [21] who report improvements in performance by orders of magnitude when application-level data management is 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-level 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 [29], SQL-Server [35], and MySQL [23] all provide security controls that apply at the level of tables, columns, or stored procedures. Oracle 10g [26] 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 parametrized view [33] of a table that is based on user-specific criteria [31]. This view essentially filters out the rows to which the current user is denied access [27].

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. As discussed in Section 2.4 Oracle 10g and IBM DB2 provide a built-in notion of lattice-ordered labels for implementing row-level security. 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 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 [34]. 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. [33] 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 fail both for diagnosis and so that users can attempt to revise their requests with the appropriate credentials to gain access. Furthermore, rather then 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, ensuring that no authorization check is missed or called incorrectly. To support this unified model of security enforcement while retaining good performance, SELinks compiles policy enforcement code to database-resident user-defined functions, which can be called directly during query processing. Database-side, as opposed to server-side, enforcement avoids the overhead of needlessly transferring inaccessible data to the server. Our experiences with two sizable web applications, a model health-care database and a secure wiki with fine-grained security policies, indicate that SELinks is flexible, relatively easy to use, and, when compared to a single-tier approach, improves throughput by nearly an order of magnitude.

Acknowledgments: We thank David An, Amol Deshpande, Jeff Foster, Justin McCann, and the anonymous reviewers for useful discussions and comments about this work. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.

8. REFERENCES

[1] S. Ambler. *Agile Database Techniques*. John Wiley and Sons, 2006.

[2] D. An. XTOLS: Cross-tier Oracle label security. Technical Report CS-TR-4934, University of Maryland, College Park, 2009.

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

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

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

[6] W.-J. Chen, I. Rytir, P. Read, and R. Odeh. DB2 security and compliance solutions for Linux, UNIX, and Windows. `http://www.redbooks.ibm.com/redbooks/pdfs/sg247555.pdf`, Mar. 2008.

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

[8] S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release, version 3.3. Located at `http://www.cs.cornell.edu/jif`, 2009.

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

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

[11] 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.

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

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

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. SOSP*, 1995.

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

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

[17] The Hop Programming Language. `http://hop.inria.fr/`.

[18] Java EE at a glance. `http://java.sun.com/javaee/`, 2008.

[19] The LINQ project. `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`, 2008.

[20] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proc. SIGMOD*, 2006.

[21] E. Müller, P. Dadam, J. Enderle, and M. Feltes. Tuning an SQL-based PDM system in a worldwide client/server environment. In *Proc. ICDE*, 2001.

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

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

[24] National Health Service. Spine. `http://www.connectingforhealth.nhs.uk/systemsandservices/spine`.

[25] OASIS XACML TC. XACML 2.0 interop scenarios. `http://docs.oasis-open.org/xacml/xacml-2.0-core-interop-draft-12-04.doc`.

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

[27] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *IEEE Symposium on Security and Privacy*, 2009.

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

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

[30] Ruby on rails. `http://www.rubyonrails.org/`, 2008.

[31] A. Rask, D. Rubin, and B. Neumann. Implementing row- and cell-level security in classified databases using SQL Server 2005. `http://www.microsoft.com/technet/prodtechnol/sql/2005/multisec.mspx`.

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

[33] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proc. SIGMOD*, 2004.

[34] V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *APPSEM-II*, pages 152–165, Mar. 2003.

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

[36] N. Swamy. *Language-based Enforcement of User-defined Security Policies*. PhD thesis, University of Maryland, College Park, August 2008.

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

[38] Volta. `http://livelabs.com/volta`, 2008.

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