Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications

Adam Chlipala
Impredicative LLC

Abstract

We present a system for sound static checking of security policies for database-backed Web applications. Our tool checks a combination of access control and information flow policies, where the policies vary based on database contents. For instance, one or more database tables may represent an access control matrix, controlling who may read or write which cells of these and other tables. Using symbolic evaluation and automated theorem-proving, our tool checks these policies statically, requiring no program annotations (beyond the policies themselves) and adding no run-time overhead. Specifications come in the form of *SQL queries as policies*: for instance, an application's confidentiality policy is a fixed set of queries, whose results provide an upper bound on what information may be released to the user. To provide user-dependent policies, we allow queries to depend on *what secrets the user knows*. We have used our prototype implementation to check several programs representative of the data-centric Web applications that are common today.

1 Introduction

Much of today's most important software exists as Web applications, and many of these applications are thin interface layers for relational databases. Real-world requirements impel developers to implement many application-specific schemes for access control ("who can do what?") and information flow ("who can learn what?"). To reason about correctness of these implementations, the programmer must consider all possible flows of control through a program.

This task is hard enough if a security policy can be expressed statically, as, for instance, a list of which of a fixed set of principals is allowed to perform each of a fixed set of actions. However, the needs of real applications tend to force use of evolving security policies, and usually the most convenient place to store a policy is in the same database where the rest of application data resides. For instance, a database often encodes some kind of access control matrix, where entries reference rows of other tables. The peculiar structure of an organization may require access control based on customized schema design and checking code. An effective security validation tool must be able to "understand" these policies.

Many program analysis and instrumentation schemes have been applied to provide some automatic assurance of security properties. In this space, the traditional dichotomy is between *dynamic* and *static* tools, based on whether checking happens at run time or compile time. The two extremes have their characteristic advantages.

- Dynamic analysis can often be implemented without requiring any *program annotations* included solely to make analysis easier.
- Real developers have an easier time writing *specifications* compatible with dynamic analysis, since these specifications can often be arbitrary code for inspecting program states.
- Static analysis can provide strong guarantees that hold for all possible program executions, even those exercising weird corner cases that may not have been considered.
- Static analysis adds no run-time overhead.

In this paper, we present a tool *UrFlow* for static analysis of database-backed Web applications. We have tried to reap some of all of the advantages just described. Our tool requires no program annotations and provides fully sound static assurance about all possible executions of a program, and it requires no changes to the run-time behavior of programs. We take advantage of the fact that it is already common for Web applications to be implemented at quite a high level, relying on an SQL engine to implement the key data structures. Our tool models the semantics of SQL faithfully, at a level that makes formal, automated analysis quite practical. We use popular ideas from symbolic execution and automated theorem-proving to build detailed models of program behavior automatically, which saves developers the trouble of explaining these models with code annotations.

It is natural for developers to write specifications that look much like the program code they are already writing. Traditional assertions (e.g., with the C `assert` macro) fall under this heading. In an application that depends on an SQL engine to manage its main data structures, it seems similarly natural to express security policies using SQL. Our tool is based on that model, allowing developers to write detailed statically-checkable specifications without learning a new language. Queries can express confidentiality properties by *selecting which information the user may learn*, and queries can express database update properties by *selecting allowable state transitions*. We need only one extension to the standard SQL syntax and semantics: to allow policies to vary by user, we introduce explicit consideration of *which secrets (e.g., passwords) the user knows*.

UrFlow is integrated with the compiler for Ur/Web [3], a domain-specific language for Web application development. Ur/Web presents a very high-level view of the domain, with explicit language support for the key elements of Web applications. For instance, the SQL interface uses an expressive type system to ensure that any code that type-checks accesses the SQL database correctly. In the present project, we have used the first-class SQL support to avoid the need for program analysis to recover a high-level view of how an application uses the database.

We begin by introducing our policy model and demonstrating its versatility. After that, we present our program analysis, including its symbolic evaluation and automated theorem-proving aspects. Next, we discuss the scope and limitations of our analysis, describe some case-study applications that we have checked with UrFlow, and compare with related work.

2 SQL Queries as Policies

Consider a simple application that maintains a database of users and per-user secret strings. We can declare our schema to Ur/Web with `table` declarations. Following standard practice in relational databases, each table includes a unique integer ID, which provides a convenient handle to pass to row-specific operations. Besides an ID, a `user` record contains a username and password, and a `secret` record contains the owning user ID and the data value.

```
table user : { Id : int, Nam : string,
               Pass : string }
```

```
table secret : { Id : int, User : int,
                 Data : string }
```

We also declare an HTTP cookie, which acts like a typed global variable which exists separately on each Web browser. This cookie tracks the authentication information for the currently logged-in user. While a more realistic program would probably rely on unique session IDs, here we adopt the less secure strategy of storing a user ID and password pair in each cookie, to simplify the example.

```
cookie login : { Id : int, Pass : string }
```

We can write a function that checks this cookie and returns its user ID if the password is correct. The code is written in a functional style, where we collapse "expressions" and "statements" into a single syntactic class. Thus, instead of determining the function return value with explicit `return` statements, we just say that the function result is the value of the single expression that is the function body.

Ur/Web code makes a lot of use of *tagged unions*, a safe analogue to C unions that is popular in functional programming languages. A tagged union value is either a simple tag, which is like an `enum` value in C; or a pairing of a tag and another value, which is like a C union, but with a convention to ensure that it is always possible to inspect a value and determine which union alternative is being used. For tag `T`, a simple tag expression is written like `T`, while the pairing of that tag with expression `e` is written `T(e)`. For instance, instead of allowing every object type to be inhabited by a special value `null`, we instead represent `null` with an explicit tag `None`, and we represent non-null object `o` as `Some(o)`. A *pattern-matching* construct `case` is used to deconstruct tagged union values.

Here is the code for a function to check the correctness of the information in the `login` cookie. It is written in a compiler intermediate language in which some higher-order functional programming idioms have been replaced with more standard imperative code.

```
fun userId() =
  case getCookie(login) of
    None => None
  | Some(li) =>
    let b = query
      (SELECT COUNT(*) > 0 AS B
       FROM user
       WHERE user.Id = {li.Id}
         AND user.Pass = {li.Pass})
      (r acc => r.B) False in
    if b then
      Some(li.Id)
    else
      error("Wrong user ID or password!")
```

Our `userId` function begins by retrieving the current value of the `login` cookie. This will either be `None`, if no value of the cookie is set; or `Some(li)`, if the ID/password record `li` has been set as the cookie value. If the cookie is not set, there is no user ID to return. Otherwise, we must consult the database to see if the password is correct.

We have literal SQL syntax embedded in the code, with splicing of variable values using curly braces. The query checks if there are any rows in the `user` table matching the cookie contents. In this intermediate language, every database read is expressed as a loop over the results of a query. The body of the loop is written as an expression with two explicitly-named new local variables: `r`, the latest row to process; and `acc`, an accumulator that is modified as we process rows. The body expression after the => determines the new accumulator value after every iteration. We give `False` as the initial accumulator value. In our example here, the loop body ignores the accumulator, and we simply project the one field of any result row to save as the accumulator. The `error` function aborts program execution with an error message, which we do here when the user provides invalid credentials.

We can write the main entry point of our application to display all of the logged-in user's secrets.

```
fun main() =
  case userId() of
    None => write("You're not logged in.")
  | Some(u) =>
    query (SELECT secret.Id, secret.Data
           FROM secret
           WHERE secret.User = {u})
      (r acc =>
        write("<li> <i>");
        write(toString(r.Secret.Id));
        write("</i>: ");
        write(escape(r.Secret.Data));
        write("</li>")) ()
```

In this query loop, the accumulator is still ignored, and in fact we execute the function body solely for its side effects, which involve writing HTML to be sent to the client.

We would like to verify that this application satisfies a reasonable confidentiality policy. Intuitively, every cell of the database belongs to a particular user. We want to ensure that no user is able to read cells belonging to a different user. This simple policy expresses our intent for the cells of the `user` table.

```
policy sendClient (SELECT *
  FROM user
  WHERE known(user.Pass))
```

The informal meaning of this policy is that the user may learn any value that could be returned from this query. Every `policy` statement is followed by a keyword naming a kind of policy. In this case, that keyword is `sendClient`, which is used for confidentiality policies. Specifically, the user may learn anything about any row of `user` *whose password he knows*. The new predicate `known` models which information the client is already aware of. We assume the client knows the text of the program and the text of the HTTP request it sent. In our example, when we disclose any secret information, we know that the user's own password is known because it came from the `login` cookie, which was part of the incoming HTTP request.

A more complicated policy allows the release of information about secrets.

```
policy sendClient (SELECT *
  FROM secret, user
  WHERE secret.User = user.Id
    AND known(user.Pass))
```

We use a join between the `secret` and `user` tables, requiring that the client demonstrate knowledge of the password for the user who owns the secret.

Our tool verifies that the application satisfies these security policies. That is, every cell of the database whose value might be disclosed could have been selected by one of these queries, based on an interpretation of `known` drawn from the HTTP request that prompted an execution.

There are several opportunities for mistakes in implementing the policy. Consider what would happen if we had implemented `userId` to always return 17. When we run the compiler, we get an error message. The compiler tells us which secret may be leaked, and (in addition to the location of the offending write) we are given a first-order logic characterization of the state of the program at the time when the leak might occur.

```
User learns: r.Secret.Data
Hypotheses: secret(x1),
  r = {Secret =
    {Id = x1.Id, Data = x1.Data}},
  x1.User = 17
```

The hypotheses are generated directly from the SQL query in `main`. The first hypothesis tells us that row `x1` is in the `secret` table. Our row variable `r` is equated with a record built by projecting the requested fields from `x1`, and the last hypothesis represents the `WHERE` clause.

In the correct implementation, UrFlow explores every static path through the program, maintaining a logical state at each point. When the analysis reaches the point that triggered the error above, we have this more informative state.

```
c = cookie/login, known(c),
c = Some(c2), user(x1),
x1.Id = c2.Id, x1.Pass = c2.Pass,
secret(x2), x2.User = c2.Id,
r = {Secret = {Id = x2.Id, Data = x2.Data}}
```

The variable `c` stands for the cookie value, which is asserted to be known to the user. The SQL query from `userId` is reflected with assertions about a variable `x1`, which is the row of `user` that must have matched the query for execution to reach this point. The confidentiality policy used a join between `secret` and `user` to describe when information on secrets may be released. The program code, on the other hand, contains no joins. UrFlow understands join semantics to the point where it is able to deduce that the above logical state implies that a join, performed as in the policy, would authorize the release of everything included in the record `r`.

2.1 What is Being Checked?

We can give a simple characterization of exactly what confidentiality property the analyzer enforces, as a function of the policy the user specifies. First, we need to define exactly what we mean by the *known* predicate. Informally, a known piece of data is something that the user is already aware of, so that no confidentiality requirement is violated by echoing back that value or another value derived from it in a predictable way. More formally, *known* is the most restrictive predicate satisfying the following rules:

1. Any constant appearing in the program text is known.
2. The initial value of every cookie is known. These cookies may have arbitrary structured types, as in the record type given to the `login` cookie in the last example.
3. The value of every explicit parameter to the application is known. For page requests generated by submission of HTML forms, this includes all form field values.
4. A record is known iff all of its fields are known.
5. For any union tag `T` (e.g., `Some` in our example), a value `v` is known iff `T(v)` is known.

We say that a value v is *allowed* in a specific database state D if there exists a `sendClient` policy that, when executed in state D, would return v as one of its outputs. We say that a value v is *built from* a set S if v is in S or can be constructed out of the elements of S by combining a subset of them with record and tagged union operations.

Now we can give a concise description of exactly what UrFlow checks. For any execution of a program that the analysis approved:

1. Whenever a `write` command sends some value v to the client, v is built from the set of values that are known or allowed.
2. Whenever the program branches based on the value v of some test expression, such that the branch chosen influences what might be sent to the client later, v is built from the set of values that are known or allowed. This prevents some *implicit flows*, where the very fact that a program reaches a particular line of code may reveal secret information. Since implicit flows are a notorious source of false alarms in information flow analysis, programmers might want to turn off this piece of checking, which would be easy to do via a compiler flag.

The same kind of characterization does not work well for ruling out implicit flows induced by SQL `WHERE` clauses, so we leave additional checking of that kind for future work. This means that a checked program may leak information about the *existence* of rows, based on tests against arbitrary SQL expressions, but the *contents* of those rows will not be leaked directly.

2.2 Authorizing Database Writes

UrFlow also checks every database modification. For example, consider this page generation function, which would be given as the action to run upon submission of an HTML form for adding a new secret.

```
fun addSecret(fields) =
  case userId() of
    None => write("You're not logged in.")
  | Some u =>
    let id = nextId() in
    dml (INSERT INTO secret (Id, User, Data)
         VALUES ({id}, {u}, {fields.Data}));
    main()
```

If we do not assert an explicit database update policy, then UrFlow rejects this program. Here is one policy that would allow the insertion:

```
policy mayInsert (SELECT *
  FROM secret AS New, user
  WHERE New.User = user.Id
    AND known(user.Pass)
    AND known(New.Data))
```

We reuse the same SQL query notation for modification policies, though the choice of `SELECT` clause is ignored, so we will always write `SELECT *`. One of the tables in the FROM clause must be given the name New; this is the table for which we are authorizing insertion.

UrFlow only allows a row insertion if the new row could be returned by one of the mayInsert queries, in a certain sense. In checking against a particular policy query, we interpret the New relation as the universal relation, containing all possible tuples. The policy may join it with other, real database tables and perform filtering with WHERE, leading to a result set of rows that may be infinite. The insertion is permitted if the New part of one of these rows matches the values being inserted.

Our insertion policy lets any user add secrets if he associates them with his own user. We can also authorize deletions and updates, based on similar criteria.

```
policy mayDelete (SELECT *
  FROM secret AS Old, user
  WHERE Old.User = user.Id
    AND known(user.Pass))
```

```
policy mayUpdate (SELECT *
  FROM secret AS Old, secret AS New, user
  WHERE Old.User = user.Id
    AND New.User = Old.User
    AND New.Id = Old.Id
    AND known(user.Pass)
    AND known(New.Data))
```

A mayDelete policy must tag a FROM table as Old, to stand for the table being deleted from. A mayUpdate policy needs both Old and New tables, standing for the part of a table being updated and the new data being written into it. Both new policies retain the logic for checking that the client knows the password for the user whose secret is affected, and the update policy also requires that the secret ID is not changed. The insertion and update policies require that the new data value is known, which provides a simple guard against inadvertent leaking of privileged information into a part of the database that is considered to be less privileged.

3 Flexibility of Query-Based Policies

We have found that this approach to writing specifications leads to natural descriptions of many natural policies. For instance, we have implemented a simple Web message forum system. Our implementation contains a table representing an access-control list. Each entry gives a user permissions in a specific forum, at a particular numeric level of access.

```
table acl : { Forum : forumId,
              User : userId, Level : int }
```

One policy allows release of information about any message in a forum that the current user has been granted any kind of access to.

```
policy sendClient (SELECT *
  FROM message, acl, user
  WHERE acl.Forum = message.Forum
    AND acl.User = user.Id
    AND known(user.Pass))
```

Posting a new message requires access at level 2 or higher.

```
policy mayInsert (SELECT *
  FROM message AS New, user, acl
  WHERE New.User = user.Id
    AND New.Forum = acl.Forum
    AND user.Id = acl.User
    AND known(user.Pass)
    AND acl.Level >= 2
    AND known(New.Subject)
    AND known(New.Body))
```

Regular users may not delete messages from forums. This right is only granted to admins, who have access level 3 or higher. The following policy formalizes the deletion rule.

```
policy mayDelete (SELECT *
  FROM message AS Old, user, acl
  WHERE Old.Forum = acl.Forum
    AND user.Id = acl.User
    AND known(user.Pass)
    AND acl.Level >= 3)
```

Our implementation allows forums to be marked as public, in which case any visitor may read their contents. There is also another ACL table which grants users admin access to all forums. Additional policies allow information flows and updates based on these rules.

The UrFlow policy language supports access control techniques besides user accounts with passwords. For example, we have implemented a simple Web-based poll system without user accounts. Anyone may create a new poll; at that time, the creator learns a secret code that grants admin rights to the poll. That code allows him to add poll questions. After adding all of the questions, the poll creator may mark the poll as live. After that time, no further changes to the poll are allowed, and the poll is added to a list on the application's front page. Anyone may vote in a live poll, but no one may vote on a poll that is not yet live. After submitting his votes, a user receives a code that allows him to view the results of the poll. Results should never be released without first checking that the user has provided a code that matches the poll admin code or a code associated with a vote that has been cast.

The policy below controls the conditions under which a new question may be added to a poll. In particular, the question must be linked to a valid poll, the user must know the admin code for the poll, and the poll must not be live yet.

```
policy mayInsert (SELECT *
  FROM question AS New, poll
  WHERE New.Poll = poll.Id
    AND known(poll.Code)
    AND NOT poll.Live
    AND known(New.Text))
```

Anyone with a poll's admin code may update the poll only to mark it as live. This policy expresses that requirement with equality assertions between old and new values of every column besides `Live`.

```
policy mayUpdate (SELECT *
  FROM poll AS New, poll AS Old
  WHERE New.Id = Old.Id
    AND New.Nam = Old.Nam
    AND New.Code = Old.Code
    AND New.Live
    AND known(Old.Code))
```

We allow release of information about answers to a poll, whenever the user proves he already voted in that poll by providing a code associated with an appropriate answer set.

```
policy sendClient (SELECT *
  FROM answer, answers AS Other,
    answers AS Self
  WHERE answer.Answers = Other.Id
    AND Other.Poll = Self.Poll
    AND known(Self.Code))
```

We believe that this specification approach is very general, while being much more accessible to the average developer than most specification languages are. To investigate the potential for static analysis based on these specifications, we implemented the UrFlow prototype, which handles a restricted subset of all SQL queries. In particular, in both policies and programs, we only process queries containing just `SELECT`, `FROM`, and `WHERE` clauses, where the `FROM` clauses must be simple comma-separated lists of tables. We also have not implemented any analysis optimizations like procedure summaries [19], and the analysis only succeeds at understanding loops and recursion following a few simple patterns.

Perhaps surprisingly, this is enough to enable sound checking of a variety of paradigmatic Web applications. We will now describe the analysis and then argue for its effectiveness with statistics about a set of representative applications that it has validated.

4 An Outline of the Analysis

Sound program checking requires considering all possible paths of execution. Since most any non-trivial Web application can effectively follow infinitely many paths, we must apply some abstraction. In implementing UrFlow, we adopted the strategy associated with tools like ESC [10], the Extended Static Checker family.

While concrete program evaluation involves program states consisting of variable values, memory states, and so on, the kind of *symbolic evaluation* that we apply involves program states consisting of *formulas of first-order logic*. Such a formula can be thought of as *describing* concrete states, so that each abstract state may stand for infinitely many concrete states. Every basic program operation can be modeled as a *predicate transformer*. Some operations may not always be safe. In the classical setting, this may be an array dereference, where the index might be out of bounds. In our case, possibly-unsafe operations include `write` commands and database updates. No matter which setting we are in, the safety of operations is checked by associating each operation with a logical condition that implies its safety.

This gives us the outline of a sound checking procedure: Start with the abstract state "true." Explore all program paths, extending the abstract state as we go. Each time we reach an operation with safety condition C while in state S, ask an *automated theorem prover* whether $S \Rightarrow C$. The ESC projects used the Simplify prover [8] for this purpose. Today, the functionality provided by Simplify is most commonly known by the name SMT, for *satisfiability modulo theories*, and there is a rich base of tools and users in the domain of static program checking.

Our outline omits a critical element of the problem: Even after abstracting program states with formulas, there are probably still infinitely many feasible program paths. The ESC approach requires additional program annotations that can be used to finitize the path space. In the design of UrFlow, we have instead taken advantage of the control-flow simplicity of the average Web application. Many interesting applications can be implemented with just one kind of loop: iteration over writing some output for every row returned by an SQL query. Such loops effect no state changes that must be taken into account in the remainder of the program, so in a sense they have trivially inferable "loop invariants." Since loop iteration does not accumulate side effects, it is sound to *traverse each loop body just once*, which ensures that each program can be broken into a finite set of finite analysis paths.

UrFlow thus works by literal exploration of all control flow paths through a program. The next section goes into more detail on the exploration strategy, pointing out the theorem prover operations that will be required. The following section presents our implementation of those prover primitives, in an engine that extends the standard SMT approach with a few new features.

5 Symbolic Evaluation

The abstract states of UrFlow are defined in terms of a simple language of logical expressions and predicates. We write c for constants (drawn from integer, floating point, and string literals), T for union tags, x for logical variables, X for program variables, F for record field names, and R for SQL table names. The following grammar describes the syntax of program states. For a token sequence t, we write $\overline{t}$ for a comma-separated list of zero or more ts.

$$\begin{array}{rrcl} \text{Expression} & e & ::= & c \mid x \mid T(e) \mid \{\overline{F = e}\} \mid e.F \\ \text{Predicate} & p & ::= & \mathsf{known}(e) \mid R(e) \mid e = e \mid \ldots \\ \text{State} & S & ::= & (\overline{p}, \overline{X \mapsto e}) \end{array}$$

A state is a pair of a variable assignment and a set of predicates. For a particular program point, a variable assignment maps every in-scope program variable into a logical expression. The predicates are expressed only in terms of logical variables, not the program variables.

Since we inline all function calls, every execution path to analyze begins at the entry point of some function that has been registered to be called in response to a particular URL pattern. The arguments to this function stand for explicit parameters and form field values, extracted from an HTTP request. Where the function arguments are named X_i, we create an initial state $(\overline{\mathsf{known}(x_i)}, \overline{X_i \mapsto x_i})$, for fresh, distinct variables x_i. At many other points in path exploration, we will generate fresh logical variables, which we always assume to be distinct from any previously-chosen variables.

For each function, we explore all paths through it. Most program expression forms are easy to process, as they admit direct translation into logical expressions. The more interesting cases come from branching and database interaction.

Our single branching construct is `case` expressions, which test a value against a number of patterns, which may bind new variables if they match. We model `if` expressions as a special case of `case` expressions, where the patterns to match against are `true` and `false`.

As an example, consider an expression like the following:

```
case e of None => e1 | Some(X) => e2
```

If `e` is just the tag `None`, then we continue with evaluating `e1`. Otherwise, `e` is `Some v` for some `v`, and we evaluate `e2` with `X` set to `v`. To capture this with symbolic evaluation, we consider both `e1` and `e2` as starts of separate execution paths. For the `e1` case, we extend the state with the predicate $v = \mathsf{None}$, where v is the result of evaluating `e`. For the `e2` case, we choose a fresh variable x, add the variable mapping $X \mapsto x$, and add the predicate $v = \mathsf{Some}(x)$.

With `case`, it is easy to write code with exponentially many control-flow paths, but where all but a few are logically impossible. For instance, we can sequence several `case` expressions that analyze the same program variable with the same patterns. Variables are immutable, so each `case` must choose the same pattern, reducing the number of feasible paths to the number of patterns. We want our automated theorem prover to detect the infeasibility of the other paths as early as possible. Concretely, this will happen on a path where two `case`s lead to assertions like $v = \mathsf{None}$ and $v = \mathsf{Some}(x)$, on a path that assumes matching of a `None` pattern the first time and a `Some` pattern the second time. The prover knows that values built with different union tags are disjoint, so it can signal a contradiction here. Whenever a contradiction is detected at some point on a path, we can skip exploring the rest of that path.

A number of primitive operations send output to the client. The simplest of these is `write`, which appends a piece of HTML to the page being generated. UrFlow enforces that the value being sent can be constructed from known and allowable pieces of data. Recall that allowable values are those that could be produced by executing `sendClient` policies in the current database state. Consider this line of our earlier example program:

```
write(escape(r.Secret.Data));
```

The record `r` has come out of a database query. To verify that this `write` conforms to the policy, we must check that `r.Secret.Data` is known, allowable, or built from such values out of record and union operations. At this point in symbolic execution, the variable mapping will map the program variable `r` to some logical variable r, and our predicate set will be:

$$\begin{array}{l} c = \mathsf{cookie/login}, \mathsf{known}(c), c = \mathsf{Some}(c'), \mathsf{user}(x_1), \\ x_1.\mathsf{Id} = c'.\mathsf{Id}, x_1.\mathsf{Pass} = c'.\mathsf{Pass}, \\ \mathsf{secret}(x_2), x_2.\mathsf{User} = c'.\mathsf{Id}, \\ r = \{\mathsf{Secret} = \{\mathsf{Id} = x_2.\mathsf{Id}, \mathsf{Data} = x_2.\mathsf{Data}\}\} \end{array}$$

The state tells us that we know of two rows that must exist in the database: x_1 from table `user` and x_2 from table `secret`. Each of our declared confidentiality policies is phrased as a `SELECT` query whose `FROM` clause mentions one or more tables. To check if a value may be written, we need to consider ways of matching the policy queries with the logical state. The same table may be mentioned multiple times in one policy or one state, so, in general, there may be many ways to match a policy's `FROM` clause with the table predicates of a state. In UrFlow, we apply the heuristic of considering at most one matching per policy. The analysis enumerates every matching of policies with row variables, subject to that constraint.

Our running example included these two policies:

```
policy sendClient (SELECT *
  FROM user
  WHERE known(user.Pass))

policy sendClient (SELECT *
  FROM secret, user
  WHERE secret.User = user.Id
    AND known(user.Pass))
```

They can be expressed in logical form, where each is a set of predicates that, if all are true, implies the allowability of a set of values.

Predicates:	$\mathsf{user}(r_1), \mathsf{known}(r_1.\mathsf{Pass})$
Values:	$r_1.\mathsf{Id}, r_1.\mathsf{Nam}, r_1.\mathsf{Pass}$
Predicates:	$\mathsf{user}(r_1), \mathsf{secret}(r_2), \mathsf{known}(r_1.\mathsf{Pass}),$ $r_2.\mathsf{User} = r_1.\mathsf{Id}$
Values:	$r_1.\mathsf{Id}, r_1.\mathsf{Nam}, r_1.\mathsf{Pass}, r_2.\mathsf{Id},$ $r_2.\mathsf{User}, r_2.\mathsf{Data}$

Matching a policy against a state is a two-step process. First, we consider a mapping of the policy's r_i row variables to variables appearing in the state. For any table predicate $R(r_i)$ appearing in the policy, we try setting r_i to x, for any $R(x)$ appearing in the state. Once we have found a plausible mapping for every policy row variable, we apply that mapping to the remaining predicates in the policy. If the theorem prover verifies that the state implies every one of these predicates, then we have found a viable policy instantiation, and we can continue matching the remaining policies. We repeat the process to try every combination of instantiating every policy at most once.

For every set of policy instantiations, we compute the set of expressions that those policies say are fair game to write. Our running example has exactly one feasible instantiation per policy: every policy variable in `user` unifies with x_1, and every policy variable in `secret` unifies with x_2. The remaining predicates are all implied by the state. Most interestingly, we must verify that the state implies $\mathsf{known}(x_1.\mathsf{Pass})$, which follows by reasoning from this subset of the state predicates:

$$\mathsf{known}(c), c = \mathsf{Some}(c'), x_1.\mathsf{Pass} = c'.\mathsf{Pass}$$

The reasoning goes like this: Because the union value c is known, its contents c' are known, too. Because the record c' is known, its field Pass is known. That field is asserted equal to the value $x_1.\mathsf{Pass}$ that we want to prove known, so we are done. The theorem prover provides a complete decision procedure for reasoning chains of this kind.

Having verified correct instantiation of each policy, we arrive at this set of allowable expressions:

$$x_1.\mathsf{Id}, x_1.\mathsf{Nam}, x_1.\mathsf{Pass}, x_2.\mathsf{Id}, x_2.\mathsf{User}, x_2.\mathsf{Data}$$

We are trying to prove that the expression $r.\mathsf{Secret}.\mathsf{Data}$ is allowable, which requires proving that it is equal to one of the above expressions. It turns out that our state implies that the written value equals $x_2.\mathsf{Data}$, because the state contains this predicate:

$$r = \{\mathsf{Secret} = \{\mathsf{Id} = x_2.\mathsf{Id}, \mathsf{Data} = x_2.\mathsf{Data}\}\}$$

That completes the check for this `write` operation. The procedure scales to handling much more complicated cases, and we also apply the same procedure to any expression used in a branching construct, such that the result of the test influences what is written to the client. Especially in this latter case, we need to be able to reason about values that are neither known nor allowable, but that are built from such values via record and union operations. Our theorem prover handles the automation of that kind of reasoning, too.

The heart of symbolic evaluation is the treatment of database queries. Recall the form of queries, as illustrated by the main output loop of our example application.

```
query (SELECT secret.Id, secret.Data
       FROM secret
       WHERE secret.User = {u})
      (r acc => ...) ()
```

We execute an SQL query, which may contain injected program values, and loop over the result rows. An accumulator is initialized to some specified value, which here is the dummy value `()`, since we execute this loop body only for side effects. Every iteration runs the loop body with `r` bound to the latest result row and `acc` bound to the current accumulator. After an iteration, the accumulator is replaced with the value of the `...` body expression.

Traditional verification tools require manual annotation of loops with invariants, to help tame the undecidability of the program analysis problem. To avoid that cost, we designed UrFlow around some observations about the loops that appear in practice in Web applications. Most are run solely for their side effects of writing content to the client, so that there is no need to track state changes from iteration to iteration. Ur/Web variables are all immutable, so it is not even possible for them to change across iterations. Side effects are restricted to database tables and cookies, which tend not to be used in the same way that variables are used in traditional imperative languages. All this implies that a simple loop traversal strategy can be very effective: *traverse each loop body only once*.

Concretely, when we reach a query in a symbolic execution path, we consider two possible sub-paths. First, the query may return no results, in which case we proceed taking the initial accumulator as the final value.

More interestingly, the loop may execute one or more times. We perform a quick linear pass over the body `...` to see which cookies it might set and which tables it might modify with SQL `UPDATE` or `DELETE` commands. All references to those cookies and tables are deleted from the symbolic state. Since all other aspects of concrete state are immutable, this new logical state is guaranteed to be an accurate description of the concrete state *at the beginning of any iteration of the loop*. Thus, by running the loop body with its local variables set to fresh logical variables, we consider all possible behaviors of the loop. We can continue execution afterward as if we had just executed the loop body once as normal, non-loop code. The symbolic state at loop exit can just as well stand for the last iteration of the loop as for any other iteration.

At the beginning of a loop iteration, we must enrich the logical state with predicates capturing the behavior of the query. This is best illustrated by example. Consider again the main loop of our example application. We execute its loop body with variable `r` set to r and `acc` set to some arbitrary value (since the accumulator is not referenced in the body). Assume that program variable `u` is mapped to logical variable u. We add these predicates to the logical state:

$$\mathsf{secret}(x_2), x_2.\mathsf{User} = u,$$
$$r = \{\mathsf{Secret} = \{\mathsf{Id} = x_2.\mathsf{Id}, \mathsf{Data} = x_2.\mathsf{Data}\}\}$$

Queries with joins just add more table predicates, as we have seen in the modeling of queries as policies. Larger `WHERE` conditions add additional non-table predicates. A `SELECT` clause determines which fields to project from the tables, in building the record expression to equate with r.

This basic algorithm works for most of the queries that we support. In general, UrFlow does not yet support SQL grouping or aggregation. We include one special case for queries selecting just the aggregate function `COUNT(*)`. Here, we consider that the loop body always iterates exactly once. Either the query result is 0, and we do not enrich the state with any new table information; or the result is greater than 0, and we assert that there exists some set of rows matching the conditions of the query.

To check database updates, we use a hybrid of the query and write checking. Any modification must match with an update policy, using the same matching procedure as for writes, but without the need to check allowability of a value. After an `UPDATE` or `DELETE`, we delete any state predicates mentioning the affected tables.

UrFlow also has basic support for simple recursive functions. Calls to recursive functions are effectively inlined like regular function calls, with further self-calls skipped. To make this omission sound, we analyze each recursive function to find all effects it might have on the database and cookies, and every self-call is treated as a nondeterministic modification of those parts of the state, followed by generation of an unknown return value. Further analysis allows us to abstract the initial state so that it can stand for any set of arguments that might be used at any recursion depth, such that we only preserve state information that can be shown not to vary across calls. As a result, just like for query loops, a single pass over the function body suffices to consider all possible behaviors.

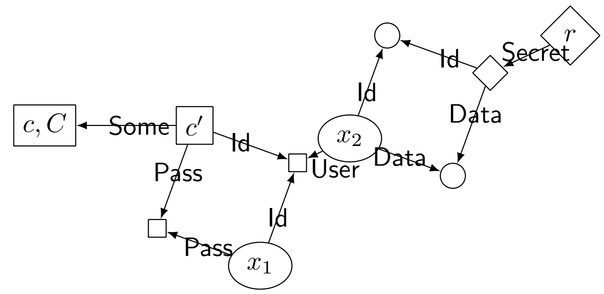

Figure 1: E-graph for the state from the `write` example

We want to emphasize some useful consequences of the way that our analysis handles SQL. First, unlike in some related work [14], despite the fact that our policies are themselves SQL queries, the analysis *does not* require that program code use exactly those queries. Semantic modeling of queries makes it possible for one policy query to justify infinitely many possible program queries. Second, the soundness of our analysis depends on knowledge of the database *schema*, but not knowledge of database *contents*. Schema changes can invalidate analysis results by, for example, redefining data integrity constraints that the theorem-prover might have relied on. However, arbitrary changes to the database rows, by arbitrary programs with no relation to UrFlow, cannot invalidate past analysis results.

6 The Theorem Prover

The last section highlighted the key theorem-prover operations that symbolic evaluation depends on. We can summarize them like this:

- Assert a predicate p. If p contradicts the predicates already asserted, raise an exception indicating so.
- Check if a predicate is implied by those already asserted.
- Determine if a logical expression can be constructed from members of a set of allowable expressions.

The first two points are supported by the classic model of first-order logic theorem-proving that is embodied in tools like Simplify [8]. The third point is new and not directly supported by usual prover interfaces, but the usual implementation techniques can support it very directly.

Provers like Simplify are based on the Nelson-Oppen architecture. We do not use many of the elements of that architecture, since our prototype implementation omits features like reasoning about arithmetic. Instead, we just adopt the key data structure, the *E-graph*. An E-graph is a directed graph representation of the possible worlds that are consistent with a set of predicates. Nodes stand for objects, and, for function symbol f, an edge labeled with f goes from node u to node v if, in any compatible world, the object associated with v equals the result of applying f to the object associated with u. A node is labeled with logical variables and constants to indicate that any compatible world must assign this node to an object equal to those variables and constants.

In UrFlow, we only use two kinds of function symbols: union tags and record field names. For tag T, there is a T-labeled edge from u to v if v must be u tagged with T (i.e., "$v = T(u)$"). For field name F, there is an F-labeled edge from u to v if u is a record whose F component equals v. For each node that came from a literal record expression, we mark that node as *complete*, in the sense that the field edges coming out of it provide a complete description of the available fields. An example of an incomplete record node is one representing a row selected in an SQL query; the state will only mention those columns relevant to the query, and it would be unsound to treat this row as if it had no further columns.

Figure 1 shows an E-graph representing the logical state given earlier for checking the code `write(escape(r.Secret.Data))`. Nodes are boxes when the state implies that they are known; other nodes may not be known. Complete record nodes are diamonds. We abbreviate `cookie/login` as `C`.

The basic prover algorithm understands two kinds of predicates: $e_1 = e_2$ and $\mathsf{known}(e)$. When either kind is asserted, its expressions are first evaluated into nodes of the E-graph, adding new nodes as necessary. A variable or constant is evaluated to the node labeled with it. A union tag application $T(e)$ is evaluated by following the T edge from the node that e evaluates to, and a field projection $e.F$ is evaluated analogously. A record expression $\{F_1 = e_1, \ldots, F_n = e_n\}$ is evaluated by checking for existing complete nodes whose F_i edges point to the nodes to which the e_is evaluate.

When a fact $e_1 = e_2$ is asserted, the nodes u_1 and u_2 standing for e_1 and e_2 are merged, taking the unions of their sets of labels and incoming and outgoing edges. Alternatively, this fact might trigger a contradiction. That happens when u_1 and u_2 are labeled with different constants or have incoming tag edges labeled with different tags.

When a fact $\mathsf{known}(e)$ is asserted, and e evaluates to u, we "change u to a box," and we propagate this knownness information across edges. That propagation follows record field edges in the forward direction only and tag edges in either direction. The same propagation is implied when merging a known node with a not-known node for an equality assertion.

The heart of the procedure is in this handling of assertion. E-graphs have nice properties which make implication checking very efficient. To check if $e_1 = e_2$, we only check if e_1 and e_2 evaluate to the same node. To check if $\mathsf{known}(e)$, we only check if e evaluates to a boxed node.

One useful addition, implemented outside of the theorem prover core, takes advantage of *key information* for SQL tables, where, for instance, an ID column is asserted not to be duplicated across rows of a table, and the SQL engine maintains this invariant with dynamic checks. Whenever a new predicate asserts that some row r is in table R, we check, for every pre-existing predicate $R(r')$, if r and r' agree on the values of R's key columns. These checks can be implemented by querying the prover core with the appropriate equality predicates. Whenever a matching r and r' pair is found, we can skip adding the new predicate $R(r)$ to the state, instead asserting $r = r'$. This enrichment of the prover is useful in analyzing applications that, for example, query a user/password table multiple times, where correctness relies on the fact that the query always returns the same result.

The last ingredient is checking if the value of expression e can be constructed out of the values of expressions $e_1, \ldots, e_n$, using only record and union operations. To implement the check, we evaluate each e_i in turn, marking its node as allowable. Next, we evaluate e to a node u. If u is marked as allowable, we are done. Otherwise, if u has an incoming union tag edge from a node v, we repeat the procedure for v. If u is a complete record node, we repeat the procedure for each target of a field edge out of u, returning success only if the check is successful for each of these new nodes. In any other case, we return failure.

7 Discussion

We can get a sense for the breadth of UrFlow by considering how it helps with the most common Web application security flaws. The OWASP Top 10 Web Application Security Risks project[1] is a popular reference for security-conscious Web developers. Based on analysis

[1] `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

of databases of real vulnerabilities, the OWASP team has identified which classes of security flaw pose the greatest risks. The Ur/Web compiler rules out injection (ranked #1) and cross-site scripting (#2) vulnerabilities and partially mitigates cross-site request forgery (#5) and unvalidated redirects and forwards (#10) using techniques unrelated to UrFlow. Risk #6, security misconfiguration, is a whole-system property that cannot really be addressed by any single tool, and UrFlow's lack of integrated reasoning about cryptography prevents it from helping to avoid insecure cryptographic storage (#7). UrFlow can contribute to the mitigation of the remaining risk categories.

Risk #3, broken authentication and session management, is helped by the ability to use UrFlow policies to specify exactly which secure tokens may be sent to which clients. It is still possible to make mistakes in the policies, but these policies should be significantly easier to audit than programs, with the many possible control-flow paths of the latter. The next two risk categories, insecure direct object references (#4) and failure to restrict URL access (#8), are very similar, as both involve the omission of access control checks for particular system objects. UrFlow can enforce that appropriate checks are always performed whenever database objects are used in particular ways. Insufficient transport layer protection (#9) could be avoided by adding a variant of `sendClient` policies which specifies values that may only be sent to clients over SSL connections.

Comparing against the pros and cons of security types [16], we find some interesting trade-offs. UrFlow uses high-level knowledge of programs to provide more sound reasoning without program annotations. Security-typed languages generally rely on declassification techniques where trust is granted to particular spans of code. This creates a contrast between the security-typed approach, requiring trusted code but granting soundness with respect to implicit flows; and the UrFlow approach, which requires no trusted Ur/Web functions but ignores some implicit flows. Security type annotations tend to be required throughout a program, while UrFlow avoids the need to mark up program code. However, SQL queries as policies involve some gotchas that would be less applicable to security types. For instance, it is easy to forget all or part of a policy `WHERE` clause, which has the unfortunate consequence of allowing behaviors by default.

The problem of implicit flow checking is a serious one in all kinds of information flow analysis. Where UrFlow checks implicit flows, the checking is not particularly clever, and implicit flows caused by `WHERE` clauses are ignored. Future work may be able to plug part of this hole statically, and we suspect there will also be a large role for dynamic monitoring systems, for detecting brute-force password cracking attempts and other attacks that involve many HTTP requests.

Many different logical languages have been used for specification-writing in static verification tools. We found SQL to be a convenient choice, because it is expressive enough to allow direct expression of interesting policies, and declarative enough to enable effective automated reasoning. We do not mean to claim that SQL has great expressivity or succinctness advantages over more traditional specification languages. Rather, most Web programmers are accustomed to SQL, which should help in overcoming some of the social obstacles faced in the past by attempts to get programmers to write logical specifications.

Our implementation today only handles a subset of the common SQL features. We omit support for outer joins. These should be easy to model via disjunctive formulas, covering all the possible cases of whether a row matching the join condition exists in a table, though a naive realization of this idea would probably have poor performance consequences for the theorem-prover. Grouping and aggregation are harder to encode in the quantifier-free first-order logic that we are employing. We suspect that most real programs can be checked with conservative encodings of aggregation, where we model aggregate function values as unknowns. Alternatively, we can restrict reasoning about aggregate functions to simple syntactic pattern-matching against policies. That approach also seems most practical for handling of the SQL `EXCEPT` operator, which implements a kind of negative reasoning about which rows do not exist. This is needed to write down policies like (for a conference management system) "reviewer A may see the reviews for paper B only if A *does not* have a conflict with B."

More advanced policies might also need to include non-trivial program code. For instance, a custom hashing or encryption scheme might be used. Here we encounter a common situation for static verification, where it is always possible to expand the reach of your theorem-prover to handle new program features. No single implementation will ever be able to handle all realistic programs, but we suspect that very good coverage will be possible, after the incorporation of significant practical experience with the tool.

8 Evaluation

The UrFlow prototype is implemented in about 2200 lines of Standard ML code. We have used the analysis to check a number of Ur/Web applications. There is a live demo of the applications, with links to syntax-highlighted source code, at:

`http://www.impredicative.com/ur/scdv/`

Application	Program (LoC)	Policies (LoC)	Check (sec)
Secret	138	24	0.02
Poll	196	50	0.035
User DB	84	8	-
Calendar	255	46	0.28
Forum	412	134	17.68
Gradebook	342	61	1.49

Figure 2: Lines-of-code breakdown in case studies, with time required to check the code with UrFlow

Our case studies include *Secret*, a minimal application for storing secrets that may later be retrieved via password authentication, which was used as the model for this paper's first set of running examples; the *Forum* and *Poll* applications from which Section 3's examples were drawn; a *Gradebook* application, for managing a database of student grades in courses; and a reimplementation of the *Calendar* application from the paper [5] that introduced the SIF system for combined static and dynamic checking of information flow in Web applications. Calendar, Forum, and Gradebook share a common user authentication component.

The Calendar application lets users save details of their schedules on the Web, with controlled sharing of information. By default, no one may learn anything about an event. The creator of an event may learn everything about it, and the creator may add invitees who inherit the same read privileges. The creator may also authorize users to know only the time of an event, so that those users see that time slot only as "busy" on the creator's calendar. Only event creators may modify any state related to their events.

The Gradebook application is based on a database of courses and assignments of users to be instructors, teaching assistants (TAs), or students in courses. Each student membership record contains an optional grade. Only system administrators may create courses and modify instructor lists. Instructors may set grades and control TA assignments. A TA may view all of the state associated with a course, but may not modify it. A student may view his own grades, and a student in a course may only affect that course's part of the database by dropping the course.

Figure 2 gives the number of lines in code in each of these components. An application's code is separated into the program itself and the policies. The figures here make "policy overhead" appear bigger than it would probably be in production applications, since our case studies include minimal code dedicated to providing fancy user interfaces. Still, these numbers compare favorably to those for systems like SIF, where Calendar requires 1779 lines of code. While we have a similar ratio of program to annotation, our annotations are of a different kind. 443 lines of the SIF version include annotations, in the form of security types [20] and explicit downgradings. The latter involve annotations that effectively say "the owner of a piece of information trusts this span of code, so let that span release derived information that would not otherwise be allowed." The SIF Calendar case study includes 17 such downgrades.

The UrFlow approach is very different. As no annotations are required in programs, there is no need to accept any part of a program as trusted. All checking is with respect to the declarative specification provided by the policy queries.

Our analysis detects flaws similar to those that occur frequently in real deployed systems. For instance, we examined reports for July 2010 in the National Vulnerability Database[2]. Among the relevant issues, we found CVE-2009-4927, involving privilege escalation via a surprising setting of a specific cookie; and CVE-2010-2685 and CVE-2009-4929, which allow administrative actions to be taken without proper credentials, via hand-crafted HTTP requests. UrFlow makes it easy to catch these problems, since it is not necessary to enumerate all possible attack vectors, thanks to policies that talk directly about underlying resources. For instance, we introduced a bug in the Gradebook application to mimic the cookie bug above, where we allow anyone to set any student's grade if a particular cookie is set to 1. The compiler complains that the database update policy may be violated, referencing the exact span of source code where the offending `UPDATE` statement occurs. The same output appears if we simulate a forgotten access control check, in the style of the second two issues above, by commenting out an important `if` test.

UrFlow also requires no change to the runtime behavior of a program, and this baseline performance level is greater than for most popular Web languages and frameworks, thanks to the general-purpose and domain-specific optimizations performed by the Ur/Web compiler. We present the performance of the UrFlow analysis itself in Figure 2, for runs on a Linux machine with dual 1 GHz AMD64 processors with 2 GB of RAM. Of our case studies, only Forum takes much longer than a second to check. This is because Forum has a complicated main function, with many security checks. Many different actions call the main function after performing some database modification. Every such call is analyzed afresh, as if the main function had been inlined. Techniques like procedure summaries [19] should make it possible to reduce this time significantly.

[2] `http://nvd.nist.gov/`

Very precise, logic-based program analyses often exhibit bad scaling behavior. There is no theoretical reason that UrFlow would not run into the same problems. Many programs with exponentially many feasible paths will indeed trigger exponential behavior in any realization of our algorithm. Simple experiments with parameterized families of programs also show that our current implementation produces exponential running time (with small constant factors) even on some examples that can probably be reduced to linear running time with more optimization. For instance, we tested programs made up of `if`-trees that perform the same SQL query at each of the tree's exponentially-many nodes. Primary key information implies that the `if` test always goes the same way, ruling out all but two paths through the tree. Still, exponential time usage results from our heuristic of considering two execution paths starting at each query, for the cases of zero or more than zero result rows. Much future work remains in smarter detection of redundant paths.

9 Related Work

The BAN logic [2] is a formal system for reasoning about knowledge in distributed system protocols. The rules of the logic model important aspects like transitive trust and cryptography. The spi calculus [1] pursues similar goals, introducing an explicit formalization of programs, rather than just of the knowledge that principals have at points throughout a protocol. Our known predicate is modeled on notions introduced in that line of work.

Security types [20] are a technique for static checking of information flow based on explicit data labels such as "high security" and "low security." The JFlow [15] and Jif [16] systems are realistic implementations of security typing for Java. SIF [5] extends Jif for the Web application domain. This line of work enables checking of a much broader range of applications than UrFlow can handle. By focusing on a narrow domain that naturally supports declarative implementation techniques, UrFlow is able to do sound checking without requiring any program annotations. Jif-based systems require many annotations, including explicit granting of trust to particular spans of code. The Swift system [4] extends this approach to do automatic, secure partitioning of Web application code across client and server, based on information-flow constraints.

Li and Zdancewic [14] presented a system for static checking of information-flow properties for database-backed Web applications. Their design requires that the application be programmed in terms of fixed sets of query templates with holes to be filled with different values on different invocations. Every template is annotated with security typing information for each input and output. In contrast, UrFlow infers the security-relevant characteristics of queries from a declarative policy. One policy may be enough to imply the sensitivity of outputs from many different query forms. UrFlow also applies theorem-proving technology to allow sound checking of more programs, including those where policies vary dynamically based on database contents.

Asbestos [9] and HiStar [23] are operating systems with support for dynamic enforcement of the Decentralized Information Flow Control model, which specifies which run-time flows between sensitive objects to allow. The Flume system [13] implements similar functionality on top of standard UNIX abstractions. All of these systems can support complex system architectures that fall outside the specialized orientation of UrFlow. Flume has been used to build a secured version of the MoinMoin wiki application. This port to Flume required about 1000 lines of new code and 1000 lines of modifications, and a performance cost between 34% and 43% was measured, against the baseline of interpreted Python code. Our Forum case study demonstrates that UrFlow can check policies based on access control lists, which are the main property enforced in the Flume case study.

The Resin system [22] implements a much lighter-weight approach to Web application security. Instead of relying on a fixed label model, Resin allows programmers to implement their own property checks in the language in which the application is written. Policy code may tag values with policy objects, and the Resin system takes care of flowing these policies through the system and checking them at points where the application interacts with its environment. Compared to the other systems we have mentioned, including UrFlow, Resin makes it much easier to add security checking to existing applications written in popular scripting languages like PHP and Python. Resin's lightweight policy approach can also express policies that UrFlow's policy queries cannot. On the other hand, once a programmer has learned Ur/Web and used it to implement his application, UrFlow requires little annotation and brings the standard benefits of static analysis, compared to Resin and the systems mentioned in the previous paragraph: we get once-and-for-all security guarantees, without the possibility of the application being aborted because a problem is detected at run-time; and we avoid extra run-time costs, such as the 33% CPU overhead reported for a representative PHP application instrumented with Resin.

Much work on Web application security focuses on *injection attacks*, where bugs allow untrusted user input to be passed to run-time program interpreters. Solutions have employed both static [12, 21] and dynamic [11, 17] analysis. Ur/Web rules out these problems by construction, by encoding the syntax of HTML and SQL with richly-typed objects.

Rizvi et al. [18] present a technique for fine-grained access control over SQL queries, based on the concept of *authorization views*, which are much like UrFlow's policy queries. The key difference is that authorization views are phrased in terms of variables like `$user-id` that must be filled in by some out-of-band mechanism. With UrFlow, the correctness of authentication may itself be verified, through reasoning about the `known` predicate. The technique of Rizvi et al. is applied dynamically to individual queries, where an allowability check against the current database must be run for each query. In contrast, UrFlow can prove statically that an application never uses query results inappropriately, with no modification to run-time database operation.

The SELinks system [7] extends the Links [6] Web programming language with support for static tracking of labels through trusted functions that enforce custom policies. The natural way of expressing some queries in SELinks involves mixing customized access control checks with code that should be compiled into SQL queries. The SELinks compiler handles the translation of the custom checks into stored procedures that the database engine can run during query evaluation. UrFlow follows the alternate approach of letting the programmer be explicit about the interaction of checks and queries, such that the static analysis verifies that all this has been done correctly. In general, SELinks provides a type system that makes certain types of security proofs easier, though the SELinks compiler does not carry out those proofs itself.

10 Conclusion

We have presented UrFlow, a static program analysis that verifies adherence of database-backed Web applications to security policies. These policies may vary by database state, and they are expressed as SQL queries, a convenient format for most Web programmers. UrFlow requires no program annotations and adds no run-time overhead. A key direction for future work is adaptation of UrFlow to more traditional languages, where database access is granted less of a first-class status, so that program analysis must be run to recover some information that UrFlow depends on.

Acknowledgements We would like to thank Stephen Chong, Avraham Shinnar, our shepherd Nickolai Zeldovich, and the anonymous referees for their very helpful suggestions about this project and its presentation here.

References

[1] ABADI, M., AND GORDON, A. D. A calculus for cryptographic protocols: The spi calculus. In *Proc. CCS* (1997).

[2] BURROWS, M., ABADI, M., AND NEEDHAM, R. A logic of authentication. *ACM Trans. Comput. Syst. 8*, 1 (1990), 18–36.

[3] CHLIPALA, A. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. PLDI* (2010).

[4] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure web applications via automatic partitioning. In *Proc. SOSP* (2007).

[5] CHONG, S., VIKRAM, K., AND MYERS, A. C. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security* (2007).

[6] COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. Links: Web programming without tiers. In *Proc. FMCO* (2006).

[7] CORCORAN, B. J., SWAMY, N., AND HICKS, M. Cross-tier, label-based security enforcement for web applications. In *Proc. SIGMOD* (2009).

[8] DETLEFS, D., NELSON, G., AND SAXE, J. B. Simplify: a theorem prover for program checking. *J. ACM 52*, 3 (2005), 365–473.

[9] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the Asbestos operating system. In *Proc. SOSP* (2005).

[10] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for Java. In *Proc. PLDI* (2002).

[11] HALFOND, W. G. J., AND ORSO, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proc. ASE* (2005).

[12] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proc. WWW '04* (2004).

[13] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *Proc. SOSP* (2007).

[14] LI, P., AND ZDANCEWIC, S. Practical information-flow control in web-based information systems. In *Proc. CSFW* (2005).

[15] MYERS, A. C. JFlow: Practical mostly-static information flow control. In *Proc. POPL* (1999).

[16] MYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java information flow, July 2001. Software release at `http://www.cs.cornell.edu/jif`.

[17] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. Automatically hardening web applications using precise tainting. In *Proc. IFIP International Information Security Conference* (2005).

[18] RIZVI, S., MENDELZON, A., SUDARSHAN, S., AND ROY, P. Extending query rewriting techniques for fine-grained access control. In *Proc. SIGMOD* (2004).

[19] SHARIR, M., AND PNUELI, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981, pp. 189–233.

[20] VOLPANO, D., AND SMITH, G. A type-based approach to program security. In *Proc. International Joint Conference on the Theory and Practice of Software Development* (1997).

[21] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *Proc. USENIX Security* (2006).

[22] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *Proc. SOSP* (2009).

[23] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proc. OSDI* (2006).