Verifying Information Confidentiality under Query Optimization in HotCRP

Richard Cho and Dan Fu

***Abstract*—HotCRP is a conference submission and review system with complex information flow policies and an expressive search capability. As a result, optimizing the search process is technically difficult and can result in information leaks if the optimization process returns either more or fewer papers than the unoptimized process. In particular, optimizations that transfer query burden across a sanitization pass can be especially problematic. In this work, we tackle this problem using formal verification. First, we develop a formal model of information flow in HotCRP. Next, we model different information flow policies and optimizations in HotCRP and use our framework to prove that the optimizations do not leak information.**

I. INTRODUCTION

HotCRP is a web-based conference submission and review system [4], [5]. One of its primary features is a strong search capability: program committee members can search for papers by title, authors, decision, and other relevant fields. With such a search capability comes a number of issues with information flow, however. For example, program committee members may themselves submit papers to the conference; in such cases, they should not be allowed to read reviews or see decisions about their paper before de-anonymization.

Such information flow issues are compounded by attempts at query optimization. In particular, information confidentiality is enforced at the level of the PHP server, but it is desirable to move query burden from the PHP server to the database. If done without care, such query optimization can result in information leakage. Consider an example of a user searching for all papers that do not have a positive "accept" decision, for example. The user should receive a list of all papers that are not written by the user and have not been accepted, and all the papers written by the user, *regardless of whether the paper has been accepted or not*. A naive optimization might move the entire query to the SQL layer and return a list of all papers that have not been accepted. By the time a server-level information policy has been applied to this list, it is too late: the user will be able to deduce which of their papers have been accepted by the absence of such papers from the returned list.

We use formal verification to address this problem by using the Coq interactive proof assistant [2] to model information flow in HotCRP and prove information confidentiality. To avoid difficulties with the source language of HotCRP (PHP), we do not attempt to prove properties of the actual PHP system. Rather, we propose a model flexible enough to capture the dynamics of information flow in HotCRP, and prove information confidentiality on number of different optimizations in that system. We leave it to the developers of HotCRP to either adopt some of the optimizations we propose or use our framework to prove the optimizations used by the actual software system.

Our framework models papers, users, policies, user queries on papers, and a simple subset of SQL. For simplicity, we model the entire HotCRP database as a list of papers, with users existing independently. We avoid the need for complex operations such as joins by simply adding relevant fields to our paper model. For example, where a real relational database might keep papers and decisions in separate tables, thus requiring joins to determine a paper's status, our model simply makes the decision a field of the paper. In this way, our framework can model rich interactions while remaining unencumbered by complex SQL logic.

Over the course of our development, we iterated over a number of policies and optimizations, from simple to complex. We also iterated over a number of proof strategies. We will present them in this paper to demonstrate the difficulty of writing correct optimizations and to help future undertakers of this or similar projects avoid common pitfalls.

In §II, we discuss relevant background about how search queries in HotCRP work. In §III, we discuss the formal model of the relevant functionality, with optimization. In §IV, we discuss the specific types of information policies we cover and the optimizations they are amenable to. In §V, we discuss our proof strategy. In §VI, we evaluate our framework and discuss the impact our proof strategy had on various proofs. In §VII, we discuss related work. Finally, in §VIII, we discuss future work and conclude.

II. BACKGROUND

The subset of HotCRP that we model consists of papers, users, and queries, all governed by an overall information policy. Papers can have a number of fields, such as a unique ID, a title, a list of authors, time of submission, etc. Similarly, users can have a number of fields and have a number of complex associations. For example, users are often attached to institutions or may play special roles in the conference, such as committee chair or conference chair. This multitude of fields engenders complex relationships and potential conflicts of interest between users and papers, and the information policy must ensure that users do not see any information that they are forbidden from.

This interaction is most difficult when dealing with HotCRP's rich query system. Users can search for papers using a number of complex query options, and queries must not leak information. Luckily, it is possible to write information policies that can prevent information leaks with some care [5]. However, the problem is complicated by HotCRP's architecture. Like many web applications, HotCRP has both a PHP server and a SQL server. It is much easier to enforce the information policy on the PHP server, since it runs on a much more expressive source language than the SQL server.

Indeed, the simplest way to ensure correctness is to load all the papers from the SQL server on the PHP server, and process the list of papers locally using the full expressive capabilities of PHP. Unfortunately, this approach has a number of problems; namely, some conferences are very large and have an extremely large number of papers. In such conferences, loading all the papers onto the PHP server may put it under extreme strain. Coupled with a large population of users all searching for results at once, this could be extremely problematic.

Thus, it is desirable to shift as much of the query burden from the PHP server to the SQL server as possible. This entails processing a user query and generating a SQL query from it to run on the SQL server. The smaller the list of papers that make it to the PHP server, the better. Unfortunately, such optimizations are non-trivial to write, and very error-prone. In this work, we propose a framework to prove such optimizations correct and demonstrate its efficacy on a family of policies and optimizations.

We make a number of simplifying assumptions to make our model simpler and make the proofs more tractable. On the SQL side, we do not model multiple tables or joins between multiple tables; rather, we simply add relevant information as fields to papers or users. We also only model a very limited subset of SQL: we can handle SELECT statements with a WHERE clause composed of field equality, AND, OR, and NOT. We collapse all information conflicts into a single team concept. Generally speaking, if a user and paper are on the same team, they have a conflict of interest (although we do allow more complex policies in later sections). Finally, we also simplify user input by modeling user queries exactly the same as SQL queries, and sharing architecture for both.

III. BASIC MODEL

We have a basic model of HotCRP where we have papers, databases, users, queries, and policies. A paper is a set of fields, namely id, title, team, and decision. A database is a list of papers. A user is a set of fields id, email, and team. A query is an inductive type that can be computed over a given paper and return true or false. It has the primitive operations of always being true or false, or testing equality of a field of a paper with a given value; this last operation requires another inductive type representing paper fields. We create more complex queries using And, Or, and Not. The Coq definitions of these objects are given in Listing 1. With these operations, queries on the database reduce to filters on a list of papers. Given a function `eval (q:query) (p:paper)` that returns true if `q` admits `p`, we can apply a query `q` to a database `db` via a filter:

```
filter (fun p => eval q p) db
```

Since we have defined a database to be a list of papers, this operation also returns a database.

However, these definitions alone are not enough to model information policies, which may scrub out fields of individual papers based on the attributes of the user making the query. For this functionality, we introduce policy maps into our model:

```
policy_map: paper -> user -> paper
```

These functions take in a paper and a user and return a sanitized version of the paper by scrubbing out individual fields of the paper based on whether the user is allowed to see them. We discuss concrete policies and the limitations we place on them in more detail in §IV-A. Given a policy `pol`, we can thus compute a sanitized version of a database `db` via Coq's `map` function:

```
map (fun p => pol p u) db.
```

Similarly to the filter function from before, this function will again return a database. Critically, this means that we can chain filter and map operations together.

```
Inductive paper : Set :=
| Paper: forall (id:nat)
  (title:string) (team:nat)
  (decision:nat), paper.

Inductive user : Set :=
| User: forall (id:nat)
  (email:string) (team: nat), user.

Notation database := (list paper).

Inductive paper_field : Set:
| Paper_id: nat -> paper_field
| Paper_title: string -> paper_field
| Paper_team: nat -> paper_field
| Paper_decision: nat -> paper_field.

Inductive query : Set :=
| True: query
| False: query
| Field_eq: paper_field -> query
| And: query -> query -> query
| Or: query -> query -> query
| Not: query -> query.
```

Listing 1. The basic model of HotCRP.

This finally allows us to define the complete operation of a user query on a database (without any optimization). Suppose a user `u` makes a query `uq` on a database `db` under policy `pol`. HotCRP must first perform a SQL query on the underlying database; in an unoptimized system, this is equivalent to a `True` query in our model. Next, the results of this query are sanitized by the policy map. Finally, the user query is applied to the sanitized list, which is returned to the user. Formally, this operates as follows:

```
filter (fun p => eval uq p)
  (map (fun p' => pol p' u)
  (filter (fun p'' => eval True p'')
  db
)).
```

IV. Policies and Optimizations

A. Policies

In order to make our proof tractable, we need to impose restrictions of the operation of policy maps. It would be impossible to optimize a randomized policy, for example. We tackled policies of varying levels of complexity throughout our project, starting from simpler ones and moving to more complex policies.

```
simple_policy (p:paper) (u:user) :=
if p.team = u.team
then (Paper p.id p.title p.team 0)
else p
```

Listing 2. The **simple** policy.

The simplest policy we have developed optimizations for is, appropriately named, **simple**. It scrubs out the decision field of any paper where the paper and the user belong to the same team (i.e., their team fields are equal). A pseudocode listing is given in Listing 2. The **simple** policy is a simple policy to start out with and facilitates reasoning about optimization strategies, but it is sorely lacking in its expressive ability. Any change in functionality requires writing a new function and accompanying proofs from scratch. To address these limitations, we have developed a model of boolean expressions, upon which we have built blacklist and whitelist policies.

First, we discuss our boolean expressions in more detail. These are very similar in structure to queries, but have a few extra fields for greater expressive power. In particular, they also include functionality for comparing user fields to particular values, and for comparing paper fields directly to user fields. These features require another type representing user fields, which are analogous to the paper fields in the query definition. These definitions are given in Listing 3. Boolean expressions are passed to a `boolean_eval` function that also takes in a paper and a user. The operation on most of the constructors is self-explanatory. On `B_paper_field` and `B_user_field`, `boolean_val` compares the value of the proper paper or user field to the value passed in during construction of the `paper_field` or `user_field` object. On `B_paper_user_field`, `boolean_eval` simply compares the given paper and user fields for equality.

With boolean expressions, we can construct families of **blacklist** and **whitelist** policies.

A **blacklist** policy is composed of an inductive type and a function that evaluates a user and a paper against that policy. The inductive type, named `b_policy` for short, takes in four boolean expressions, one for each field of a paper. Given an instantiation of a `b_policy`, a paper, and a user, the blacklist function evaluates each boolean expression for the paper and the user, and scrubs out the corresponding field of the paper if the expression evaluates to `true`.

```
Inductive user_field : Set :=
| User_id: nat -> user_field
| User_email: string -> user_field
| User_team: nat -> user_field.

Inductive boolean_exp : Set :=
| B_true: boolean_exp
| B_false: boolean_exp
| B_paper_field: paper_field ->
  boolean_exp
| B_user_field: user_field ->
  boolean_exp
| B_paper_user_field: paper_field ->
  user_field -> boolean_exp
| B_and: boolean_exp -> boolean_exp ->
  boolean_exp
| B_or: boolean_exp -> boolean_exp ->
  boolean_exp
| B_not: boolean_exp -> boolean_exp.
```

Listing 3. Boolean expressions.

The **whitelist** policy is structured the same way, except it only scrubs out the corresopnding field of the paper if the expression evaluates to `false`. Definitions of the inductive types and pseudocode for the blacklist function are given in Listing 4. With these definitions, we can define general families of policies and prove statements about the entire family of policies. The only limitation is that conditions for policy scrubbing must be expressible in terms of boolean expressions on user and paper fields.

B. Optimizations

An optimization is defined to be a function that maps a user query to an inner and outer user query. The inner query represents the optimized SQL query, and the outer query represents the PHP processing that occurs post-sanitization. In a correct optimization, for all databases db, users u, policies P, and user queries uq, a paper is in the set of papers accepted by uq on db scrubbed by P if and only if it is in the set of papers accepted by the outer query applied to the set of papers obtained by using the policy to scrub the set of papers accepted by the inner query.[1]

We created two optimization functions for the simple policy, one which just relaxed the user sql by replacing every instance of `Paper.decision` with `True`. The first method treats the policy scrubbers as black boxes and then modifies the user query to replace any fields that

[1]For a different expression of this, see §V.

```
Inductive b_policy : Set :=
| B_policy:
  forall (id_exp:boolean_exp)
    (title_exp:boolean_exp)
    (team_exp:boolean_exp)
    (decision_exp:boolean_exp),
  b_policy.

Inductive w_policy : Set :=
| W_policy:
  forall (id_exp:boolean_exp)
    (title_exp:boolean_exp)
    (team_exp:boolean_exp)
    (decision_exp:boolean_exp),
  w_policy.

b_policy_map (pol:b_policy) (p:paper)
  (u:user) :=
Paper
  (if b_eval pol.id_exp p u
  then 0 else p.id)
  (if b_eval pol.title_exp p u
  then "" else p.id)
  (if b_eval pol.team_exp p u
  then 0 else p.team)
  (if b_eval pol.deciesion_exp p u
  then 0 else p.decision)
```

Listing 4. Blacklist and whitelist inductive types and the blacklist policy map function.

are scrubbed out with `True`. In the simple policy, this is the decision field. This effectively relaxes the user query by preventing it from looking at any information that gets scrubbed by the policy and is unsafe to look at before the policy is applied. This method generates an inner query that accepts a paper if the original query accepts a paper after the policy has been applied. The outer query in this case is the same as the original query, since this will guarantee that the optimization is correct. While this method is capable of handling arbitrarily complex policies, a simple relaxment function can suffer from relaxing the query too much until the inner query just becomes effectively an `True`. This proof was far easier due to not having to look inside the boolean expressions inside the policies to show it correct.

The secondary method takes into account the construction of the policy blacklist to move the entire user query into a query that can be applied before the policy that accepts papers if and only if it is accepted by the original user query after the policy has been applied. This is possible because our policy map function as previously defined has a boolean expression that can be written in terms of a user query because the boolean expression can only look at paper fields, user fields (which are constants at query evaluation) and only support And, Or, and Not logic. We call this function `bexp_to_query`. With this, we can then rewrite the user query to account for the policy.

We will integrate the "if (boolean expression) then (scrub with 0) else (original value)" in `b_policy_map` as a into the user query expression using Or, And, and Negation. Given a user query uq, and a policy that scrubs out $field$ if $bool\_exp$ is true, we can rewrite every occurence of $field$ inside the uq with:

```
if field = 0 then
  Or bexp_to_query(bool_exp) uq
else
  And not(bexp_to_query(bool_exp)) uq
```

After applying this for every field that the policy scrubs, we now have an transformed user query that accounts for the policy and can be run before the policy, making it a suitable inner query. We then prove that this inner query only accepts papers that would have been accepted by the user query on a paper scrubbed by the policy. The outer query is then just the `True` statement, so it returns everything passed in to it. This is fine since our inner query has a very strict paper acceptance guarantee. These together make up our optimization, which we proved correct.

V. Proof Strategy

We want to prove that our optimizations do not let leak any information by showing extra fields or papers that should not be included or hiding any fields or papers that should have been shown. The strongest statement of correctness for an optimization might be expressed as follows: let `u` be a user,`uq` her query, `pol` the policy, and `db` the database. Furthermore, let `map_pol` be a function that takes in a list of papers and applies `pol` to each paper, and `filter` be a function that takes in a query and a list of papers and filters them by `uq`. Then an `(opt_outer, opt_inner)` pair is correct iff:

```
forall u uq pol db,
  filter(uq,
    map_pol(filter(True, db))) =
  filter(opt_outer(uq, u),
    map_pol(filter(opt_inner(uq, u),
    db)).
```

That being said, this statement is actually stronger than we need for correctness in this domain. In particular, we only care about list membership, not order. As a result, we instead prove the following weaker result, defined using Coq's `In` function:

```
forall u uq pol db p,
  In p filter(uq,
    map_pol(filter(True, db))) <->
  In p filter(opt_outer(uq, u),
    map_pol(filter(opt_inner(uq, u),
    db)).
```

Please see the footnote.[2]

```
forall u uq pol p,
  eval uq (pol p u) =
  eval (opt_inner pol uq u) p u &&
  eval (opt_outer pol uq u) (pol p u).
```

Listing 5. Simplified statement of correctness for optimizations.

A. Layering Strategy

Our final lemma, as stated, is difficult to work with. The optimization functions are nested below layers of filter and map calls. Luckily, we can prove a simpler statement that goes a long way to proving the membership inclusion version of correctness. The statement is shown in Listing 5. The left hand side of the statement represents the unoptimized version: every paper is fetched from the database, then the policy is applied to it, and then the user query is applied to it. The user only sees the paper if the user query admits the sanitized version of it. The right hand side represents the two stages of the optimized query. First, the inner SQL query must admit the raw paper. Next, the outer query must admit the sanitized version of the paper. If both sides of this statement are true for any paper, then we know that any paper that the optimized query shows the user was also shown by the unoptimized version, and vice-versa.[3]

B. Queries

During our initial optimization tests, we found that not having an explicit Not made generating the optimization and proving it correct much easier, particularly for the

[2]Alas, we derped. This condition is actually not strong enough to prove information confidentiality; in particular, the number of occurrences is actually important. We didn't realize this until the wee hours of Tuesday morning, by which time it was too late ☹. (See the next footnote for why this isn't so bad).

[3]In some ways, this is actually a stronger statement than the original thing that we wanted to prove. We're pretty sure that we can prove the original equality statement from this statement, but we just ran out of time.

relaxing optimization. This is because the use of a Not can invert the "relaxment" replacement which introduces further complexity that isn't needed. In order to safely get rid of Not while preserving the possible complexity of the original query language we must introduce a field not equals type. This is because if there was a Not, we can use DeMorgan's law to push the Not all the way down to the ends of the query, where it will hit a `True|False|Field_eq|Field_neq` which can be written as their negated counterparts, removing the Not.

We then proved our optimizations on the Not-less queries, and then showed that the definition of a query without a Not is equivalent to the definition of a query with a Not.

C. Policies

We employed a similar simplification strategy for the proof of our **blacklist** and **whitelist** policies. We found that the inclusion of the `Not` constructor in the boolean expressions made inductions difficult. As a result, we introduced simplified boolean expressions that used `Paper_field_neq`, `User_field_neq`, and `Paper_user_field_neq` constructors. We proved the **blacklist** optimization correct on this family of simplified boolean expressions, and then wrote a function to convert from simplified boolean expressions to our original boolean expressions. After proving that translation correct, we were able to easily extend our previous correctness proofs to the policies with the original boolean expressions.

Once we had that in place, we were almost immediately able to prove correct a set of **whitelist** policies as well. A **whitelist** policy is just the opposite of a **blacklist** policy; in particular, one can convert a **whitelist** policy to a **blacklist** policy by just negating each clause of the **whitelist** policy. With a generalized boolean expression language, this is easy: the translation simply applies the `Not` constructor to each boolean expression. We can thus save a great deal of effort by simply translating **whitelist** policies to **blacklist** policies and applying the **blacklist** optimizations.

VI. Evaluation

A. Experience

Our simplification strategies greatly reduced the amount of proof effort necessary to prove our optimizations correct. We can say this with some degree of anecdotal evidence, since we did not use our layering strategy when proving our optimizations correct for the **simple** strategy. As a result, although the **simple** strategy is significantly simpler than either the **blacklist** or **whitelist** policies, it took considerably more effort and man-hours to prove our optimizations correct than did the later optimizations with the layering strategy in place.

We also have anecdotal evidence that removing the `Not` constructor greatly simplified the proof process and made it much more tractable. In particular, one author spent roughly eight hours trying to force a proof of the **True** optimization on the **simple** policy before removing the `Not` constructor. After he removed the `Not` constructor and proved the optimization correct on the simplified query language, the other author was able to write a translation and prove the optimizations on the original queries correct with less than an hour of work and roughly three lines of proof code.

Overall, our entire development consists of $2,291$ lines of Coq code (not including 112 lines of graveyard where we stored incorrect optimizations). Our proofs of the optimization for the **simple** policy took roughly 27 man-hours to prove correct, and the final proof is 703 lines long, including helper lemmas. Our proofs of the simplified **blacklist** policy took roughly 6 man-hours, and the final proof is 464 lines long. The proofs of the generalized **blacklist** and **whitelist** policies took roughly 2 man-hours, and the proofs are 66 lines long.

B. Incorrect Optimizations

Part of the reason why the optimizations for the **simple** policy were so hard to prove is that it is in general difficult to write a correct optimization function, even with access to the author of HotCRP. In this section, we provide some examples of incorrect optimizations that we caught during the proof process.

In early iterations of our second optimization, we did not realize the need for different replacements of field equality/inequality based on what the user searched for. In our first attempt at optimization, we tried replacing every instance of `paper.decision = x` with `(paper.team = u.team) || (paper.decision = x && paper.team != u.team))`. This optimization is incorrect if $x \neq 0$. In this case, the user should not see any paper from their own team, since the policy will scrub the decision field, and `paper.decision = x` will evaluate to `false`. Under this optimization, the user will see all papers from their team.

We tried rectifying this incorrect optimization with what turned out to be another incorrect optimization. This time, we tried replacing every instance of `paper.decision = x` with `(paper.team`

`= u.team && paper.decision = 0) || (paper.decision = x && paper.team != u.team))`. Again, this will show the user all of their papers with decision 0, even if they ask for $x \neq 0$.

Our third incorrect optimization corrected this error, but introduced more. We replaced every instance of `paper.decision = x` with `(paper.team = u.team && x = 0) || (paper.decision = x && paper.team != u.team))`. This is the closest that we came to a correct optimization before finally arriving at the correct version presented earlier in this paper. This latest optimization fails because if the user searched for papers with decision 0, they will not see any papers that do not belong to their team.

Later during the implementation of the correct spec, we ran into another bug in the field comparison function from user fields to paper fields. In comparing fields, the type of the paper field being compared to was lost via the use of the unserscore operator, which led to an unprovable state. However, with a small change and explicitly matching every case, the correct field matching function was written and the proof went though.

C. Limitations

Although our **blacklist** and **whitelist** definitions are powerful, there are some policies that they cannot capture. For example, they cannot capture policies that depend on arbitrary (deterministic) computation. There is no way to express an inequality such as `paper.team > x`, for example, or `paper.title.length > 10`. It is certainly possible to write policies and try to prove optimizations about them in our framework; one simply has to write a map function in Coq. However, it will be difficult writing optimizations that can successfully move entire user queries into the SQL layer with policies such as these. Certainly our second optimization will be unable to do so. It is possible that our first optimization could work, but we did not have the time to try this.

VII. Related Work

Another approach tackles security in HotCRP with policy agnostic programming [7]. The problems in HotCRP that were brought up were problems such as indirect leaks (users executing a series of commands to deduce information that they do not have privilege too directly access), incorrect viewers (resolving permissions for the wrong user) and policy spaghetti (policy code getting mixed into other code which makes tracking information flow annoying). The paper resolved the problems by tracking information flow policies across application code and database queries with their policy agnostic framework and Jacqueline, which works with out of box relational databases. Our work differs in that we don't provide a policy agnostic framework on which to create applications with a smaller trusted computing base. We care about a very specific problem, which is the optimization of queries across the application code and database boundary (since the policy is enforced in application code but the query is a database query). We then show, given a spec of queries, policies, papers, and users, that the optimizations are correct; the optimization function doesn't leak any information.

Our model of SQL is very simple. Richer models of relational databases may yield more expressive policies and more powerful optimizations. Bezaken et. al. have proposed a Coq formulation of the relational database model and have used it to prove a number of theorems about relational databases. Such a formalization might be beneficial for our approach.

Our framework also does not make any claims about the actual running HotCRP system. The system is written in PHP and relies on a fairly massive trusted computing base. There has been some work in proving parts of this trusted base correct, however. Malecha et. al. have built a lightweight, verified relational database system in Coq [6]. This system is much less feature-complete than many conventional enterprise database systems, but the subset of SQL that we model is simple enough that we could run our framework on top of this system. On the other end, Filaretti and Maffeis have developed an executable formal semantics for a core of PHP. Using these semantics, one could potentially prove some functionality on the PHP side of HotCRP correct.

VIII. Future Work

Although our development and framework have allowed us to prove optimizations correct on a large family of policies, our work is by no means complete. In the immediate future, we would like to prove strict list equality, as stated in $\S V$. We believe this is eminently possible given the strength of the statements that we have already proven. We would also like to extend our blacklist and whitelist policies to allow hybrid combinations of blacklist and whitelist conditions. It should be fairly straightforward to prove optimizations about these by translating the hybrid lists into straight blacklists.

In the longer term, we would like to extend the blacklist conditions past boolean expressions into the realm of general computation. With robust intermediate theorems about which fields a general computation depends on, along with stronger use of the inner/outer optimization divide, we believe that we can create significant optimizations for policies that incorporate general computation.

Our development is also not very closely tied to any concrete instatiation of HotCRP. While we have proposed optimizations and proved them correct, we do not know whether these optimizations or similar variants are employed by the actual HotCRP system. Thus, even though we have proven our optimizations correct, the actual system might theoretically be leaking information left and right.[4] In the future, we would like to either implement our policies in the native HotCRP or prove HotCRP's actual optimizations correct in our framework. In a similar vein, we would like to write export functions to export functionality from our framework to PHP or SQL. This is another way we could ensure that the code running in the HotCRP system is correct.

Another issue that we would like to tackle in the future is whether these optimizations actually result in better performance. The overriding assumption through our entire project has been that moving query burden to the SQL server is beneficial. However, we have seen that, for some optimizations, this results in more complex queries than the original user queries. It is thus an open question whether the tradeoff in more complex queries is worth moving the query burden to the SQL server.

Finally, and perhaps most importantly, we would like to name our system. A few promising candidates have come to mind–HotCRP+Coq, HotCRP In Coq, CoqCRP, and HotCoq, among others. Of these, we feel that CoqCRP most accurately captures the engineering quality of our development. However, we feel that it is slightly misleading, given that we have not actually written a CRP system in Coq. We feel that such a system could be an interesting direction for future work, however.

HotCRP In Coq is also promising, since it almost describes what we did, i.e. write a formal model of parts of HotCRP in Coq. However, we again feel that name to be misleading, since we did not model the entirety of HotCRP in Coq. We run into a similar issue with the name HotCoq. The Hot in HotCRP comes from the Hot Topics in Networks conference; alas, we have reproduced neither this conference nor anything about networks in our development. [5] We are left with HotCRP+Coq, which, to be frank, is just not as interesting as the other options. In lieu of a better option, we have simply elected to leave our framework unnamed for the time being.

References

[1] V. Benzaken, É.Contejean, S.Dumbrava. A Coq Formalization of the Relational Data Model. In: Shao Z. (eds) Programming Languages and Systems. ESOP 2014.

[2] https://coq.inria.fr/.

[3] D. Filaretti and S. Maffeis. An executable formal semantics of PHP. Published in *ECOOP'14 European Conference on Object-Oriented Programming*, 2014.

[4] E. Kohler. HotCRP. https://hotcrp.com/.

[5] E. Kohler. Hot Crap! Published in *WOWCS'08 Proceedings of the conference on Organizing Workshops, Conferences, and Symposia for Computer Systems*, April 2008.

[6] G. Malecha, G. Morrisett, A. Shinnar, R. Wisnesky. Published in *POPL'10 Principles of Programming Languages*, January 2009.

[7] J. Yang. Preventing Information Leaks with Policy-Agnostic Programming. PhD Thesis, MIT. September 2015.

[4]That being said, we are pretty sure it isn't...