Security Types That Work: Type Driven Repair for Database-backed Web Applications

Jordan Brown
School of Computer Science
Carnegie Mellon University
jmbrown@andrew.cmu.edu

***Abstract*—Time has proven that programmers are not good at thinking about information leaks. This is not a surprising fact; in modern applications, information flows through potentially many different pieces of functionality, each with their own security permissions, before finally being displayed to a user. Database interactions make this problem even harder, as programmers now have to reason about policy enforcement across the different semantics of application code and database queries. In this paper, we introduce Binah, a web framework that takes responsibility for managing information flow policies across the application and database. Binah supports a policy-agnostic programming model, where policies are attached directly to sensitive values instead of littered throughout the code. This allows the programmer to centralize all of the security policies for their application into one place, making it easier for them to focus on the core application logic, and making it easier to audit and modify their security policies. We evaluate the expressiveness of Binah policies in our ongoing implementation and verification of the SIGBOVIK conference management system.**

I. Introduction

As we store more and more information online, it seems that large scale information leaks are becoming more frequent. Even Facebook, a company with entire teams dedicated to protecting user privacy, was unable to prevent a subtle information leak, which lead to the discovery of James Comey's private Twitter account. In this case, a reporter sent a friend request to one of Comey's family members on Instagram, and an algorithm that suggests friends revealed an account that belonged to Comey. This lead the reporter to the discovery of Comey's Twitter account.

The engineers at Facebook are not completely at fault here. Information flow is an inherently difficult problem for programmers to think about, as it requires placing security checks in every instance where a sensitive value is used. This quickly becomes tedious and obfuscates the true functionality of the underlying program. However, when more than a billion people trust Facebook to protect their private data, the difficulty of this task is no longer relevant.

There have been many attempts to solve the problem of enforcing information flow policies at the language level, ranging from runtime solutions as presented in Labeled IO [4] to static solutions as in Ur/Web [2]. The problem we solve can be described with four sub-problems. 1. How can we enforce information flow security policies? 2. How can we do so in database-backed web applications while 3. eliminating both run time overhead and 4. the need for the programmer to manually fix instances where information flow policies are broken? Every solution so far only covers at most three of these sub-problems.

In this paper, we introduce a practical framework for policy agnostic programming that we call Binah. Binah allows its users to attach policies directly to sensitive values, holding the machine, rather than the programmer, responsible for enforcing them. As a result, policy agnostic programming provides strong static guarantees, more readable application code, and a much easier way to audit security policies. Since Binah works statically, there is no runtime overhead introduced past correctly enforcing the policies. Binah is built on top of Yesod, a web framework for Haskell, and uses Liquid Haskell [5] to verify expressive properties across both the application and database.

In this paper, we show our contributions in pushing the work done in Lifty [1] to a real web framework using a SQL database. In addition, we introduce the work we have begun on policy inference.

II. Verification

We will use an example of a conference management system to show Binah in action and help the reader develop an intuition for what policy agnostic programming can look like in Haskell.

Binah makes use of Liquid Haskell, an expressive and decidable dependent type system for Haskell, to express its policies. A type in Liquid Haskell is a Haskell type refined with a boolean predicate. For example, a natural number can easily be expressed in Liquid Haskell as:

```
{-@ type Nat :: {v:Int | v >= 0} @-}
```

Using liquid types, we can also start to write basic policies. Consider the following function signature, which should only reveal the reviewers of a certain paper to a program chair:

```
{-@ getReviewersForPaper ::
          {u:User | isPc u}
          -> Paper
          -> [{u:User | isReviewer u}] @-}
getReviewersForPaper :: User
                     -> Paper
                     -> [User]
getReviewersForPaper u p = ...
```

If `getReviewersForPaper` is called with a user that is not checked to satisfy the isPc property, Liquid Haskell will throw a compile time error. Notice, too, that the return type is refined to express that every user returned in this list is a reviewer. This function queries the database, and Liquid Haskell can now use properties from the query to describe the objects returned.

More interestingly, we can add liquid refinements to database queries that return values satisfying real world relationships. This can be slightly more complicated, as many-to-many relationships between two models are expressed using a third table in SQL databases. This third table would not appear in the function signature, which makes it slightly more tricky to refine. For example, consider a query that gets the list of reviewers who must review a certain paper:

```
{-@ measure reviewer :: User
                     -> Paper
                     -> Bool @-}
{-@ assume isReviewer :: u:User
           -> p:Paper
           -> {b:Bool | reviewer u p} @-}
isReviewer :: User -> Paper -> Bool
isReviewer u p = ...

{-@ getPapersForReviewer :: u:User
       -> [{p:Paper | reviewer u p}] @-}
getPapersForReview :: User -> [Paper]
getPapersForReviewer u = ...
```

In this case, we must introduce a helper function, `isReviewer`, and a Liquid Haskell measure, `reviewer`, to refine the query. Not all functions may be referenced in the refinement logic, so we use measures in Binah to express that certain properties hold true that we cannot directly express in the logic. We assume the helper function to be true, since it is expressing a real world relationship that cannot be verified by Binah on its own.

This is all well and good, but this still only enforces access control. To enforce information flow, we need to make sure that the policies used when the data is accessed are also enforced wherever the data is used. This is a common way for information to leak, and indeed the way James Comey's Instagram account was found. Although the algorithm that suggests friends was allowed to see information that would have linked James Comey to his son's Instagram account, the data ended up flowing to the reporter, who did not have permission to see that data. To enforce information flow, we leverage Liquid Haskell's abstract refinements [6] and propagate these policies as abstract refinements throughout our program.

We admit that the above examples were simplified in order to make the examples more understandable. In Yesod, almost every function written is wrapped in the `Handler` monad. The `Handler` monad in Yesod is equivalent to the controller in the common model-view-controller pattern. In Lifty, policies were propagated using `Tagged`, which was the Lifty equivalent of the Haskell `Identity` monad, but with added refinement types. In order to reduce the amount of programmer effort, we refined the `Handler` monad directly. Now, let's revisit the function where we were fetching the reviewers for a paper:

```
{-@ getReviewersForPaper ::
    {u:User | isPc u}
    -> Paper
    -> Handler <{\u -> isPc u}>
       [{u:User | isReviewer u}] @-}

getReviewersForPaper :: User
                     -> Paper
                     -> [User]
getReviewersForPaper u p = ...
```

Notice the addition of `\u -> isPc u`. The policy is a function from `User` to `Bool`. Here, we say that only the Program Chair can see the reviewers on any arbitrary paper.

While `Handler` is used to propagate the policies, we can now ensure they are enforced at every output sink. Suppose we have the following output function liquid signature:

```
{-@ safeShow :: Show a => Handler<p> a
                       -> User<p>
                       -> Text @-}
```

This type expresses a function that takes a `Handler` carrying data of type a with a policy p, a user u that satisfies $p\ u$, and returns `Text`. In order to get Binah's static guarantees, any output sink must take in a user, just as `safeShow` does. If there is a case that the policy on the `Handler` is not satisfied by the User's refinement type, Liquid Haskell throws an error.

III. Inference

Binah capitalizes on Liquid Haskell's type inference to infer policies already implemented in the application in order to minimize the amount of effort a programmer has to put in to use Binah. In Yesod, there is a function named `isAuthorized` that is called to determine whether or not a page can be viewed. Implementations of this function usually check properties of the currently logged-in user and relationships between that user and the parameters of the request. `isAuthorized` returns `Authorized` if a user may access a page, and `Unauthorized` if the user may not access the page. Take, for example, the following function, which only allows users who are the program chair to access the `ProgramChairR` route:

```
isAuthorized :: Route
             -> Handler AuthResult
isAuthorized ProgramChairR = do
   user <- getCurrentUser
   return (case isPc user of
      True -> Authorized
      False -> Unauthorized)
```

This is a very good place to infer policies for a given route. From this function, we can infer that when we are

responding to a request to the `ProgramChairR` route, the current user must be the program chair. We can then propagate inferred policies like these to the endpoints `isAuthorized` is protecting. In Yesod, each request has a specific Haskell function that is called to respond. Suppose the function that responds to the `ProgramChairR` route calls `getReviewersForPaper` as we defined it above. If `isAuthorized` checks to make sure the currently logged-in User is indeed the program chair, there is no reason to throw an error for not checking if the User is the program chair here again. Unfortunately, inferring policies is not a trivial task. While Liquid Haskell does provide type inference, it will not try to create new liquid types to infer, since the space of potential liquid types is infinite. Instead of relying completely on Liquid Haskell as is, we have to augment it with a preprocessing phase that will infer liquid types that we find meaningful from the code itself.

IV. Guarantees and Limitations

Binah benefits from the same soundness guarantees presented in Lifty– users will never be shown data they are not allowed to see. However, our solution is not complete. Since we approach the problem statically, we conservatively enforce information flow policies. As a result, there are instances where users will not be shown data they are in fact allowed to see. Consider the following example from Polikarpova's talk on Lifty [7]:

```
sendStatusUpdate :: ()
sendStatusUpdate = do
  papers <- getAllPapers
  pcMembers <- getProgramCommitteeMembers
  sendEmail pcMembers papers
```

Here, we need to insert checks to make sure that members of the program committee do not receive status updates on papers for which they are marked as conflicts. The repaired code would then check to make sure that only papers for which the conflicts and the `pcMembers` were disjoint were actually sent. This can lead to users in `pcMembers` not seeing papers that they are not marked as conflicts for– instead of creating an email for each `pcMember`, an email that would be appropriate for every `pcMember` would be created.

In general, when sensitive data is flowing to multiple output sinks, we take the conjunction of the policies to retain soundness. This can result in overly strict policies being enforced. We address potential solutions to this problem in Future Work.

V. Evaluation of Usability

To evaluate the usability of Binah, we wrote a conference management system for SIGBOVIK and began verifying expressive policies regarding the database interactions. We found that encoding policies in Liquid Haskell has a bit of a learning curve, which will restrict the feasibility of our system for the average Haskell programmer. As well, we found that we could not easily write self-referential policies in Liquid Haskell. For example, consider the policy where a user should be able to see the authors of a paper if they themselves are an author on that paper. We would express that with the following refined database query:

```
{-@ viewAuthors ::
    u:User
    -> Handler<{\u -> user in users}>
       users:[User] @-}
```

Since the policy appears before the data being queried, the variable `users` is not in scope. Although this is inconvenient, it seems to be more of an engineering problem than a fundamental limitation of Liquid Haskell.

Other than self referential policies, we did not find any policy that we could not encode into Liquid Haskell's refinement type system. Of course, though, since Liquid Haskell's refinement logic is decidable, the expressiveness is somewhat limited.

VI. Surprises and Lessons Learned

Throughout the process of working on Binah, the main recurring lesson I learned was that the openness of problems in research leads to a lot of time spent thinking about what the actual problem I want to solve should be. Perhaps equally as important, I learned that the results presented in papers are very rarely the first effort at solving a problem– research involves lots of trial and error, and I definitely encountered a lot of failure.

Another important lesson that I learned is that research tools are no more robust than any other piece of software. There can be pages and pages of inference rules and soundness proofs, but the implementation will always have bugs somewhere. I am grateful to the Liquid Haskell team for all the patience they had with me, especially every time I asked for a new feature.

VII. Related Work

Verifying security properties of web applications has seen great improvements in the last decade. UrFlow is a static analysis tool built for Ur/Web, a domain specific language for web applications. UrFlow can soundly and statically check information flow policies in web applications, and infers policies from SQL queries in the user's code. In Jeeves [3], Yang et al. support a policy agnostic programming model for Python that is enforced dynamically. While the dynamic approach worked, in practice it introduced too much additional overhead to be practical for real web applications. In Labeled IO, Stefan et al. dynamically verify security policies in Haskell using labels, but suffer from the same performance overheads as Jeeves. The work presented in this paper is the logical next step for Lifty, a language for policy agnostic programming. Lifty allows users to specify security policies at the sources of sensitive data, and automatically propagates and enforces those policies everywhere the data flows. Lifty accomplishes this using refinement types and program synthesis where verification fails. Though we have not yet ported the synthesis tool from Lifty to work with Haskell, synthesis will play the

same role in our solution. However, Lifty does not have a database model, and as a result is not usable for real web applications.

VIII. Future Work

This work is just the beginning of a much longer line of research, and there are many problems we would like to address going forward. First, a tool must be implemented to automate the policy annotating process. A promising solution to this problem is to generate safe database functions for the user from the policies they provide. We plan to also implement a preprocessing phase in which policies are inferred from existing application code. Furthermore, instead of conservatively enforcing information flow, we plan to fall back to dynamic enforcement of the policies in cases where we would be too conservative in order to minimize the behavioral changes in correct code. Additionally, we plan on simplifying the policy encoding process, making it easier for users to write and test their policies. A possible concrete avenue we would like to explore is generating policies from test cases, and creating a policy testing framework in Binah.

IX. Conclusion

We present a web framework that supports a policy agnostic programming model. Specifically, we introduce a way to encode expressive policies within Liquid Haskell's refinement type system. Currently, this web framework guarantees that no user will see data they are not authorized to see, but may over-approximate precise information flow. We've found that the usability of the type system requires more learning than may be appropriate, so we have identified the key future step of inferring policies to minimize the amount of annotations required.

References

[1] Polikarpova, Nadia, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. "Type-Driven Repair for Information Flow Security." arXiv preprint arXiv:1607.03445 (2016).

[2] Chlipala, Adam, and L. L. C. Impredicative. "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications." In OSDI, pp. 105-118. 2010.

[3] Yang, Jean, Kuat Yessenov, and Armando Solar-Lezama. "A language for automatically enforcing privacy policies." In ACM SIGPLAN Notices, vol. 47, no. 1, pp. 85-96. ACM, 2012.

[4] Stefan, Deian, Alejandro Russo, John C. Mitchell, and David Mazires. Flexible dynamic information flow control in Haskell. Vol. 46, no. 12. ACM, 2011.

[5] Vazou, Niki, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. "Refinement types for Haskell." In ACM SIGPLAN Notices, vol. 49, no. 9, pp. 269-282. ACM, 2014.

[6] Vazou, Niki, Patrick M. Rondon, and Ranjit Jhala. "Abstract refinement types." In European Symposium on Programming, pp. 209-228. Springer Berlin Heidelberg, 2013.

[7] Polikarpova, Nadia. " Type-Driven Program Synthesis." Lecture, Pittsburgh, PA, March 29, 2017.