# Fine-grained, Language-based Access Control for Database-backed Applications

A THESIS PRESENTED<br>
BY

Ezra B. Zigmond

TO<br>
The Department of Computer Science

IN PARTIAL FULFILLMENT OF THE HONORS REQUIREMENTS<br>
FOR THE DEGREE OF<br>
Bachelor of Arts<br>
IN THE SUBJECT OF<br>
Computer Science

Harvard University
Cambridge, Massachusetts
March 2018

Thesis advisor: Professor Stephen Chong Ezra B. Zigmond

# Fine-grained, Language-based Access Control for Database-backed Applications

## Abstract

Database-backed applications often run queries with more authority than necessary. Since programs can access more data than they legitimately need, flaws in security checks at the application level can enable malicious or buggy code to view or modify data in violation of intended access control policies. Although database management systems provide tools for controlling access to data, these tools are not well-suited for modern web applications which often have many users and consist of many different software components. First, databases are unaware of application users, and creating a new database user for each application user is impractical for applications with many users. Second, different components of the same application require different levels of database access, which would require creating different database users for different software components. Thus, it is difficult to properly limit the authority an application has when executing queries.

I propose ShillDB, a language for writing secure, database-backed applications. ShillDB enables reasoning about database access at the language level through capabilities, which limit what database tables a program can access, and contracts, which limit what operations a program can perform on those tables. ShillDB contracts are expressed as part of function interfaces, making it easy to specify different access control policies for different components of an application. These contracts act as executable security documentation for consumers of ShillDB programs and are enforced by the language runtime. Further, ShillDB provides database access control guarantees independent of the security mechanisms of the underlying database management system.

I have implemented a prototype of ShillDB and have used it to implement the backend for a lending library reservation system. My experience indicates that ShillDB is a practical language for enforcing database access control policies in realistic, multi-user applications and has reasonable performance overhead.

# Contents

# Acknowledgments

First, I would like to thank my advisor Prof. Stephen Chong. I am grateful to him for inspiring my interest in programming languages research, welcoming me into his research group, and providing encouragement and guidance as I worked on this thesis.

I am especially grateful to Prof. Christos Dimoulas for mentoring me, arming me with the knowledge to write my own Racket contracts and macros, and providing feedback on the design of ShillDB. I wish him the best in his new professorship!

I would also like to thank Prof. Margo Seltzer and Prof. James Mickens not only for generously agreeing to be my thesis readers, but also for teaching fun and challenging courses that were among the highlights of my undergraduate experience.

Thanks to Scott Moore for providing invaluable guidance during the early stages of this project. Thanks to Kevin Loughlin and Adam Ehrenberg for helping me refine my ideas and clarify my writing. Thanks to Gwen Howard for helping me design a poster version of this research and for encouraging me to go rock climbing instead of writing all day. Thanks to Luca Schroeder for being a constant colloborator the past four years and for reminding me that if you think you're hot stuff, you're probably a hot mess.

Finally, I am grateful to my parents for their support and for fostering my interest in computers from a young age, even though they couldn't understand why I wanted so many different books on programming.

# 1

# Introduction

Database-backed applications often require dynamic, fine-grained access control in order to secure sensitive information. For example, a multi-user web application such as Facebook must restrict the information that it displays based on the currently logged-in user. These access control policies are often enforced at runtime by specially written security code which sits between the database management system (DBMS) and the rest of the application code. This security code may validate user inputs, filter out rows returned by queries, or be embedded in the SQL queries the application issues. However, such security code can be difficult to write correctly and must be modified any time security policies change. Any bugs in the security policy code can result in unauthorized disclosure of information: in 2015, a security researcher found that it was possible to access other users' private photos on Facebook due to an access control bug [1]. These sorts of access control vulnerabilities are both serious and common: OWASP ranks broken access control as one of the ten most critical security risks to web applications [3].

| students | | | |
|---|---|---|---|
| *id* | *name* | *email* | *gpa* |
| 1 | Mike Birbiglia | birbigs@college.edu | 2.5 |
| 2 | Tig Notaro | tnotaro@college.edu | 3.9 |
| 3 | Patton Oswalt | poswalt@college.edu | 3.4 |

| advising | |
|---|---|
| *student* | *advisor* |
| 1 | Jerome Seinfeld |
| 2 | Jerome Seinfeld |
| 3 | Joan Rivers |

**Figure 1.1:** Schema and example data used by a student directory application.

Applications are typically run with more privilege than needed, which contributes to the prevalence of broken access control. In the most extreme cases, this can take the form of running a server as `root` or accessing a database as an administrator. If a malicious user is able to find vulnerabilities in the application's input validation or access control checks, she can execute commands with the authority of these privileged user credentials. To limit the possible damage from such an exploit, good practice dictates that a program ought to execute with just the authority it needs to perform its functionality. This concept is known as the Principle of Least Privilege (POLP) [33].

Unfortunately, many systems do not adequately support running programs with least privilege. Moore et al. [27] identify two primary obstacles to following the POLP in most systems. First, it is difficult to determine what authority a piece of software legitimately needs to execute. Second, mechanisms to limit authority are difficult to use due to being too coarse-grained or requiring significant changes to software. These problems hold true for running a database-backed application with least privilege. As an example, consider a student directory application for use by professors at a university. Suppose the application is backed by data from two database tables: a table `students` that stores student records and a table `advising` that maps student ids to the name of the student's advisor (Figure 1.1). It is fine for any professor to view non-sensitive information about students (for example, their name and email), but suppose that university policy requires that a professor should only be able to view grade-point averages for her own advisees.

According to the POLP, the student directory application *should not even be able* to access information that the logged-in user should not view according to policy. However, running the application with least privilege presents several difficulties. The first difficulty is determining what privilege the application needs to perform its legitimate function. In practice, this would require reading through all of the code and examining the database queries the program makes. Once one has inferred the authority that the directory application needs, it may or may not be possible to correctly limit the application's database access, as some DBMSs (e.g. MySQL) do not provide the sort of tuple-level access control mechanisms necessary to express the access control policy described. Further, implementing user-based access control at the DBMS level remains cumbersome because each application-level user would need to map to a distinct database user.[*] Additionally, different components of the application require different levels of database access for their legitimate purpose. For example, a component of the directory application that lets a professor send an email to all her advisees has no legitimate need to access students grades. Certain application components may also be less trusted than others (perhaps they were written by an untrusted third-party), and one may reasonably want to run these components with less privilege than the rest of the application. Giving components different privilege levels using DBMS-level access control would require creating even more database users. Finally, specifying application-specific access control policies in the DBMS separates the security policy from the program, making it difficult to understand the application's access control policies just by reading the application source code.

There is therefore a need for a more pragmatic approach to expressing and enforcing database access control policies in multi-user applications consisting of many different components. To address this need, I propose ShillDB: a language with support for declarative security policies that describe and limit a program's database access. Consumers of ShillDB programs can examine the programs' security policies to ascertain the authority with which the program will run. In this way, the policies serve as executable security documentation.

At the core of ShillDB is CapQL, a new database interface I have developed that operates on *view capabilities*. These capabilities are unforgeable tokens that represent access to a view of a database table (or tables) and confer the authority to perform database operations on that view (such as fetching or updating data). Much as database views have been used at the DBMS level to

[*] While this may be possible for a student directory application at a small university, it is not practical to create a new database user for every person who uses Facebook, for example.

```
1  (provide
2    [display-students
3     (->i/join
4     ([X (user)
5         #:post (lambda (v)
6                  (where v (sqlformat "student = id AND advisor = $1" user)))
7         #:with (view/c +where +fetch)])
8     ([user string?])
9     [(view/c +join +fetch) #:groups X]
10    [(view/c +join
11             [+fetch #:restrict (lambda (v)
12                                  (select v "name, email"))]) #:groups X]
13    any)])
14
15 (define (display-students user v-advising v-students)
16   #| Application code goes here... |#)
```

**Figure 1.2:** ShillDB contract for a student directory application. The language runtime will enforce the specified access control policy regardless of the implementation of the `display-students` function, and the policy is cleanly separated from the rest of the application code.

implement content-based access control [4], view capabilities provide ShillDB programs with restricted windows into database tables. In ShillDB, view capabilities can be received only as initial program arguments (or derived from existing capabilities), providing a basis for reasoning about a program's database access. Further, ShillDB enables programmers to write *contracts* on functions which state and restrict how functions can use capabilities they receives. Thus, it is possible to deduce the authority a ShillDB program will have at runtime just by examining what capabilities are passed in and what contracts will be applied to the capabilities.

Figure 1.2 shows a possible ShillDB contract for the student directory application described above. The contract is a security specification for the function `display-students` defined on lines 15-16. The function takes three arguments: the name of the currently logged-in user (`user`), a view of the advising table (`v-advising`), and a view of the students table (`v-students`). The **`provide`** form (lines 1-13) takes a function and a contract and exports the function with the given contract applied. The contract for `display-students` (lines 3-13) specifies a contract on the arguments and the return value: `user` must be a string (line 8), the two view arguments are restricted using *view contracts* (lines 9-12), and the function can return any value (line 13).

ShillDB's `view/c` form can be used to specify contracts on view capabilities. A contract on a view consists of a list of *privileges*. Privileges specify what operations can be performed on a view: for example, line 9 specifies that `v-advising` can be joined with other views (`+join`) and that the view's underlying data can be fetched (`+fetch`). Privileges can have *modifiers* to provide finer-grained policies: lines 11-12 specify that the `+fetch` privilege for `v-students` can only be used to fetch the `name` and `email` columns.

ShillDB contracts can also specify *join groups* which restrict how view capabilities can be joined together. Lines 4-7 define a join group `X` which restricts that if two views in the group are joined together, the join must be an equijoin on the `student` and `id` columns and the resulting view must be restricted to just the logged-in user's advisees (lines 5-6). The result of the join derives an unmodified `+fetch` privilege (line 7). The contracts on the two view arguments (`v-advising` and `v-students`) specify that the views are in the join group `X` (lines 9 and 12). The contract is defined using the `->i/join` *contract combinator* which can be used to define function contracts that involve join groups where the group's definition depends on the value of a function argument (in this case, `user`).

This contract both documents the privilege that the directory application needs to run and provides guarantees about what database access the program has, regardless of the implementation of the `display-students` function. In particular, users are restricted in which students' GPAs they can view because the only way to receive a `+fetch` privilege to view the `gpa` column in the `v-students` view is by joining it with `v-advising` and restricting the resulting view to just the logged-in user's advisees. Further, since none of the view contracts specify `+update`, `+delete`, or `+insert` privileges, there is no way for the application to modify the underlying data in the views. Finally, observe that the contract is cleanly separated from the implementation of the application, making it easy to read or modify the contract without understanding or changing the implementation details.

ShillDB offers language features for reasoning about and restricting programs' access to databases. Specifically, ShillDB uses a new, capability-based interface for accessing databases and provides contracts for restricting the use of database capabilities. These features make it possible to run database-backed applications while following the POLP without needing to use DBMS-level security tools.

The rest of this thesis is structured as follows. Chapter 2 provides necessary background on behavioral contracts, capability-based security, and using contracts to restrict the use of capabilities. Chapter 3 introduces a capability-based API for accessing relational databases. Chapter 4 presents the design and implementation of ShillDB. Chapter 5 reports on the usability and performance overhead of ShillDB. Chapter 6 examines related work. Chapter 7 concludes and proposes directions for future work.

# 2

# Background and Prior Work

This chapter covers background knowledge necessary to understand the contribution of this thesis. Section 2.1 introduces higher-order behavioral contracts with an emphasis on their use in the Racket programming language. Section 2.2 overviews capability-based security and capability safety. Section 2.3 illustrates how behavioral contracts and capabilities can be used together to express and enforce security policies.

## 2.1 Behavioral Contracts: Motivation and Introduction

Consider a Racket [15] function that removes a substring from a string (Figure 2.1). The function takes three arguments: a string (`s`), an index into the string (`start`), and the length (`len`) of the substring to remove. Informally, the behavior of the function is that it should return a new string that is the same as input string, except that the substring starting at index `start` of length `len` has been removed:

```
(define (string-excise s start len)
    (string-append (substring s 0 start)
                   (substring s (+ start len) (string-length s))))
```

**Figure 2.1:** A Racket function for removing a portion of a string. This simple implementation does not enforce many aspects of the function's specification (e.g. argument types).

```
(: string-excise (-> String Integer Integer String))
(define (string-excise s start len)
    (string-append (substring s 0 start)
                   (substring s (+ start len) (string-length s))))
```

**Figure 2.2:** `string-excise` function from Figure 2.1 written in Typed Racket with a type annotation. This implementation will reject obviously incorrect programs that pass in function arguments with the wrong types.

```
> (string-excise "Contracts are no fun!" 13 3)
"Contracts are fun!"
```

Implicit in the informal specification of the function are several invariants. Most clearly, the data types of each argument and of the return value are specified: the first argument must be a string, the second two arguments must be integers, and the function must return a string. A static type system can enforce this part of the specification and statically reject certain obviously wrong programs (e.g. programs that reverse the argument order). This is easily possible using Typed Racket [40], a dialect of Racket that has support for function type annotations and static type checking (Figure 2.2).

Consider, however, a less obvious invariant of `string-excise`: the excised substring should not extend past the end of the input string (that is, `(+ start len)` must be less than the length of the string). Traditional static type systems such as those found in C, Java, ML, or the current release of Typed Racket* are not expressive enough to write this constraint as part the function's type signature. Since semantic properties of programs cannot be decided statically in general, programming languages that support dependent types (types defined in terms of a value) or static verification of program properties often require programmers to write proof annotations (e.g. Microsoft's Dafny [18]) or even full proofs (e.g. Idris [2] or Coq [17]). Thus, there is a tradeoff between a type system's expressiveness and its usability.

Even if this invariant on the function's arguments cannot be written as part of the type signature, it is still necessary to handle invalid input more gracefully than the function currently does. Calling the function with arguments that violate the invariant on the length of the excised substring results in an error message that might be confusing to users of the function:

```
> (string-excise "Contracts are no fun!" 13 10)

 substring: starting index is out of range
  starting index: 23
  valid range: [0, 21]
  string: "Contracts are no fun!"
```

Someone calling `string-excise` may not know (and should not need to know) that the function uses `substring` internally. In order to debug this error, a user may need look at the implementation details of `string-excise`. Ideally, `string-excise` should provide an error message at a more appropriate level of abstraction. One way to accomplish this is to add special error-checking code into the function body. The example in Figure 2.3 makes clear the downsides of this approach to invariant checking: what began as a simple, three-line function is now a mess of error-handling code that spans 18 lines. Further, the invariants on the arguments are still not obvious from the signature of the function. For this particular function, it is easy enough to read through the implementation and figure out the restrictions; however, in a more complex function, it could be difficult for a consumer of the function to determine the specification, especially if error-checking code is interspersed with the main function logic.

*Behavioral contracts* provide an alternative approach for expressing and enforcing specifications. Behavioral contracts allow attaching pre- and post-conditions to function interfaces by writing code in the same language as the implementation. These conditions are executable code and are checked

[*] Typed Racket recently added experimental support for type refinement and dependent function types that can statically check linear integer arithmetic properties with no user proof annotations [32]. This functionality is sufficient for the simple invariants of the `string-excise` function, but not for specifications that involve more complex properties.

```
(: string-excise (-> String Integer Integer String))
(define (string-excise s start len)
  (unless (and (>= start 0) (< start (string-length s)))
    (raise-arguments-error
     'string-excise
     "start index is out of range"
     "start index" start
     "valid range" (format "[0, ~a)" (string-length s))
     "string" s))
  (unless (and (>= len 0) (< len (- (string-length s) start)))
    (raise-arguments-error
     'string-excise
     "excised substring extends past string end"
     "start index" start
     "substring length" len
     "valid lengths" (format "[0, ~a)" (- (string-length s) start))
     "string" s))
  (string-append (substring s 0 start)
                 (substring s (+ start len) (string-length s))))
```

**Figure 2.3:** Checking invariants in `string-excise`. Understanding the specification of the function requires reading through the error-handling code in its body.

```
1 (define/contract (string-excise s start len)
2   (->i ([s string?]
3         [start (s) (and/c integer? (>=/c 0)
4                           (</c (string-length s)))]
5         [len (s start) (and/c integer? (>=/c 0)
6                               (<=/c (- (string-length s) start)))])
7        [result string?])
8   (string-append (substring s 0 start)
9                  (substring s (+ start len) (string-length s))))
```

**Figure 2.4:** Using a contract to check invariants in `string-excise`. The function's specification is concise and cleanly separated from the implementation.

dynamically (that is, at runtime). Contracts thus let a user write complex specifications using familiar syntax without requiring the specifications be statically-checkable.

Figure 2.4 shows an implementation of `string-excise` that uses Racket's **`define/contract`** form to attach a contract to the function. The contract itself is on lines 2-7. The contract provides an executable specification for the invariants that must hold for each argument and the result: the first argument must be a string (line 2), the second argument must be a valid index in the string (lines 3-4), the last argument must be a valid length that does not cause the excised substring to extend past the end of the input string (lines 5-6), and the result must be a string (line 7).

**`->i`** is a *contract combinator* provided by Racket that takes in contracts on the function arguments and the return value and returns a contract for the function. The combinator allows creating *dependent contracts* wherein the contract on an argument depends on the values of one or more other arguments. For example, the contract on `start` depends on the value of `s` (line 3) since the valid indices into the string depend on the length of the string.

The contract on each of the arguments is a *flat contract* (a predicate function that can be checked on a value immediately) built out of simple contract combinators. For example, `>=/c` (lines 3 and 5) is a contract combinator that takes in a number $n$ and returns a contract requiring that a value be $\geq n$. `and/c` (lines 3 and 5) is a contract combinator that takes any number of contracts and returns a contract that requires a value to satisfy all of the given contracts.

With the contracted version of `string-excise`, providing the same invalid `len` argument results in a descriptive (albeit verbose) error message:

```
> (string-excise "Contracts are no fun!" 13 10)
 string-excise: contract violation
  expected: (and/c integer? not/c negative? (<=/c 8))
  given: 10
  in: the len argument of
    (->i
     ([s string?]
      [start (s) (and/c integer? (>=/c 0) (</c (string-length s)))]
      [len (s start) (and/c integer? (>=/c 0)
          (<=/c (- (string-length s) start)))])
     [result string?])
  contract from: (function string-excise)
  blaming: program
  (assuming the contract is correct)
  at: program:1.18
```

The contract-based version has three distinct advantages over the previous implementation. First, the specification is cleanly separated from the implementation. As a result, it is easy for a consumer of the function to read and understand the specification without looking at the implementation. This separation also makes it easier to change either the specification or the implementation without modifying the other. Second, the contract automatically generates detailed error messages (similar to those provided by the error-checking version in Figure 2.3) without additional work from the function author. Finally, the contract tracks *blame*, which means that it determines which component is at fault if the contract is violated. In the example above, it is clear that the caller of `string-excise` is to blame for supplying an invalid argument. One could also imagine a case in which the implementation of `string-excise` was incorrect and the function broke its own contract by returning something other than a string. Blame information can be useful for debugging, especially when a program involves complex interactions between different software components.

So far, this example has primarily considered flat contracts: contracts which can be checked immediately when applied to a value. Suppose, however, that one wishes to attach a contract to a higher-

```
(define/contract int-map f lst
  (-> (-> integer? integer?) (listof integer?) (listof integer?))
  (map f lst))
```

**Figure 2.5:** Attaching a contract to Racket's `map` function. In general, this sort of higher-order contract cannot be checked immediately when the function is applied.

order function like Racket's `map` (which maps a function over a list). Figure 2.5 defines a function `int-map` which requires that the given function (`f`) maps integers to integers and that the input list (`lst`) contains integers. The contract also ensures that the output list contains integers. **`->`** is a contract combinator that creates a simple function contract. The last argument to **`->`** is the contract on the function result, and the other arguments are the contracts on the function arguments. The contract on the input list and the output list, `(listof integer?)`, is a flat contract; it is easy to check that a list contains only integers as the list flows into or out of the function.

But how can Racket check the contract on `f`? Racket cannot check a contract on a function simply by examining the body of a function because statically checking semantic properties of functions is undecidable in general. Instead, Racket wraps the function in a proxy called a *monitor* that will check the contract whenever `f` is applied and when it returns. The argument contract is checked when a value flows into `f` and the result contract is checked on values that `f` returns (in this case, both contracts are `integer?`). Thus, in the example below, the contract on `int-map` is not violated even though the provided function *could* return a non-integer value if a list with even elements were passed as the `lst` argument:

```
> (int-map (lambda (x) (if (odd? x) (+ x 1) "not an integer")) '(1 3 5))
'(2 4 6)
```

The contract on `int-map` is a *higher-order contract* because it is a function contract defined in terms of another function contract. In addition to complicating contract checking, higher-order contracts also complicate assigning blame correctly. Dimoulas et al. define blame as correct if a party is only blamed when they control the flow of values into the contract check that failed [10]. Considering this criterion, if the body of `int-map` passed a non-integer argument to `f` in violation of the contract, one must blame the implementation of `int-map` since it controls the values passed into `f`. On the other hand, if `f` returned a non-integer value in the body of `int-map`, the supplier of `f` is to blame since they control the behavior of `f`. The contract monitor is responsible for correctly tracking blame and providing errors in the case of contract violations.

Although contracts allow writing and enforcing rich specifications that could not be checked statically, contracts are not unilaterally preferable to static types. First, since contracts are checked at runtime, contracts incur an overhead during program execution (whereas types can be checked during compilation and then erased from the executable code). Second, types provide static assurances about program properties, while contracts provide no guarantees about what *could* go wrong, instead only providing errors if something *does* go wrong. In certain settings (e.g. aerospace or medical software), having static guarantees may be extremely important.

## 2.2 Capability-based Security

A *capability* is an unforgeable token that both designates a resource and conveys the authority to perform some action(s) on that resource. In *capability-based security*, capabilities are the only means to access certain resources (that is, a subject's access to resources is determined based on what capabilities they possess). It is useful to contrast capability-based security with the more familiar notion of access control lists (ACLs). ACLs are familiar if one has run `chmod` or examined the output of `ls -l`. In a system with ACLs, resources store information about what access different subjects have to that resource. For example, an operating system associates with each file information about which users can write to that file. When a subject wishes to access a resource, they refer to the resource by some well-known name (such as a file path) and specify the action they wish to perform. This sort of authority, wherein subjects can request to access resources by name, is called *ambient authority*.

Considering the difference between capabilities and ACLs (or ambient authority more generally) helps make clear properties of capabilities that make them well-suited for supporting the POLP. For example, when a program runs, it typically inherits the ambient authority of the invoking user, meaning that the program can modify almost any of the user's files. If file system access were instead mediated by capabilities, it would be possible to control what files a program could access by controlling what capabilities the program is given. Another useful property of capability-based systems is that they allow for easy dynamic subject creation [25]. It is easy to create a new capability that confers limited access to a resource, while in an ACL-based system, this would typically involve creating new users or user groups. The unwieldiness of subject creation in ACL-based systems can lead to coarse-grained permissions and failure to follow the POLP.

In *object-capability languages*, every reference to an object is treated as a capability to access that object. This means, for example, that object references cannot be forged, in contrast to languages such as C++ where a program can forge a reference just by creating a pointer that contains an object's location in memory. In an object-capability language, there are a limited number of ways for an object to acquire a reference to another object [23]:

1. Initial conditions: Two objects may reference each other before the program runs.
2. Parenthood: When an object creates a new object, the parent holds the only reference to the child.
3. Endowment: An object can close over objects in the environment in which it is defined.
4. Introduction: An object can receive objects as arguments to its methods or as return values from methods it invokes.

These restrictions on how object references can be acquired are sometimes called *capability-safety*, and thus object-capability languages are known as *capability-safe languages*. Capability-safety provides a basis for reasoning about what capabilities different components of a program may access. In a capability-safe language, it is also possible for one component to grant another component restricted access to a capability it possesses by wrapping the original capability in a proxy object before sharing it. This proxy object intercepts any requests to the underlying capability and can choose how to handle the request (for example, passing it on to the proxied capability or rejecting it). Using proxy components, one can enforce complex access control policies like revocable access [25]. The

```
(define carkey%
  (class object%
    (super-new)

    (define/public (unlock-car)
      #| Unlock the car ... |#)))

(send (new carkey%) unlock-car)
```

**Figure 2.6:** Implementation of a `carkey` class and an example of instantiating a car key object and invoking a method.

next section demonstrates how these sorts of capability design patterns can be implemented using behavioral contracts and the advantages of doing so.

## 2.3 Contracts and Capabilities for Security: An Example

Suppose one is implementing software for Zapcar, a new car-sharing company that allows customers to rent cars for short periods. One might implement access to the rental cars by controlling access to car key capabilities. Figure 2.6 shows a Racket implementation of a simple `carkey` class[†], where a car key has a method `unlock-car` which will unlock the car to which the key refers. The details of the unlock implementation and how car keys are initially created to refer to particular cars are not important for this example.

Although users should be granted a car key capability at the start their rental, this capability must be revocable so that users cannot keep accessing the car after their rental expires. This security policy can be implemented using Redell's "Caretaker" pattern, which uses a forwarding receiver object to allow a capability to be revoked [31] (Figure 2.7)[‡]. `makeCaretaker` takes in a `carkey` object and returns two values: a `caretaker` and a `gate` which both close over a mutable boolean, `enabled?`.

[†] Racket is not a capability-safe language (for example, Racket supports object serialization, which makes it possible to obtain object references by reading from the file system), so it is not accurate to refer to Racket objects as capabilities. This consideration is separate from the goal of demonstrating how to implement capability design patterns using contracts, and so examples are shown in Racket to focus on features of Racket's contract system.

```
(define (makeCaretaker carkey)
  (let* ([enabled? #t]
         [caretaker (new (class object%
                           (init)
                           (super-new)
                           (define/public (unlock-car)
                             (if enabled?
                                 (send carkey unlock-car)
                                 (error "capability revoked")))))]
         [gate (new (class object%
                      (init)
                      (super-new)
                      (define/public (enable)
                        (set! enabled? #t))
                      (define/public (disable)
                        (set! enabled? #f))))])
    (values caretaker gate)))
```

**Figure 2.7:** Implementing the caretaker pattern for `carkey` instances using a proxy object. The caretaker can intercept any requests to the underlying car key capability and forward or reject the request based on the value of `enabled`.

When the `caretaker` object receives the `unlock-car` method, it will invoke the method on the underlying car key object only if `enabled?` is true. The value of `enabled?` can be manipulated by invoking the `enable` and `disable` methods on the `gate` object:

```
(define ck (new carkey%))
(define-values (caretaker gate) (makeCaretaker ck))

> (send caretaker unlock-car)
> (send gate disable)
> (send caretaker unlock-car)

 capability revoked
```

[‡] The code in Figure 2.7 and Figure 2.8 is adapted from Moore's [28] discussion of contracts on capabilities.

```
(define (makeCaretakerContract)
  (let* ([enabled? #t]
         [caretaker/c (object/c (unlock-car (->*m () #:pre enabled? void?)))]
         [gate #| ... same as before ... |#])
    (values caretaker/c gate)))
```

**Figure 2.8:** Implementing the caretaker pattern for `carkey` instances using a contract. The implementation of the gate is the same as in Figure 2.7 and elided here for brevity.

To provide a car key with revocable access, a component can share the caretaker object rather than the original `carkey` capability.

An alternative approach is to implement the pattern using Racket's contract system (Figure 2.8). The function `makeCaretakerContract`, when invoked, returns a contract and a `gate`. Instead of creating a proxy object, this implementation returns a contract constructed using the **`object/c`** combinator, which takes contracts on fields and methods and returns a contract that can be applied to an object. The **`->*m`** combinator is a variant of **`->`** specialized for methods and which allows adding a boolean-valued pre-condition to a contract. When `enabled?` is true, the precondition in the method contract will pass, but when `enabled?` is false, a contract violation will be signaled and the method invocation will fail:

```
(define-values
  (caretaker/c gate) (makeCaretakerContract))

(define/contract ck
  caretaker/c
  (new carkey))

> (send caretaker unlock-car)
> (send gate disable)
> (send caretaker unlock-car)
```

```
 unlock-car: contract violation
  #:pre condition
  in: the unlock-car method in
```

```
   (object/c
    (unlock-car (->*m () #:pre ... void?)))
  contract from: (definition ck)
  contract on: ck
  blaming: program
   (assuming the contract is correct)
  at: program:1.17
```

The advantages of using the contract approach over creating a proxy object are twofold. First, because of the features of Racket's contract system, the contract implementation is more concise while also providing more detailed error messages, including blame information. Second, the contract can be used enforce the security specification at function *interfaces* rather than having to do so in function implementations. With the contract approach, one can easily write a contract on a function that will make sure that the return value is wrapped in a caretaker:

```
(define-values
    (caretaker/c gate) (makeCaretakerContract))

(define/contract (get-carkey)
    #| ... some implementation |#)

(provide (contract-out [get-carkey (-> caretaker/c)]))
```

Here, the program exports a function `get-carkey` using **`provide`** and attaches a contract to the exported function ensuring that the result of the function will be wrapped in a caretaker. In this case, it is easy to look at the contract on `get-carkey` and verify that the security policy will be enforced on `carkey` instances returned by the function. By contrast, when using the proxy object approach, one would have to read through the implementation of `get-carkey` (and potentially functions that `get-carkey` calls) to make sure that the return value is appropriately wrapped in a caretaker.

In this example, using contracts both provided a convenient way to implement a common capability design pattern and pushed the work of enforcing security policies from the implementation of a function to its interface. Chapter 4 discusses how ShillDB uses contracts on capabilities to express and enforce database access control policies.

# 3

# CapQL: A Capability-based Database Interface

Running database-backed applications with least privilege requires designing a database interface that makes it easy to reason about what tables (and what rows and columns of those tables) a program can access. Typical SQL-based interfaces make this sort of reasoning difficult for two primary reasons. First, queries can refer directly to any tables that the user executing the query has access to, making it difficult to know what tables and rows a program could access without reading through every query the program might issue. Second, queries often conflate different operations into one `SELECT` statement: a single statement may perform many operations like joining tables, selecting rows, and aggregating data. A SQL-based interface therefore does not enable writing fine-grained security policies using the same ontology that the queries use.

An interface designed for running database-backed applications with least privilege should make

| students | | | |
|---|---|---|---|
| *id* | *name* | *email* | *gpa* |
| 1 | Mike Birbiglia | birbigs@college.edu | 2.5 |
| 2 | Tig Notaro | tnotaro@college.edu | 3.9 |
| 3 | Patton Oswalt | poswalt@college.edu | 3.4 |

| advising | |
|---|---|
| *student* | *advisor* |
| 1 | Jerome Seinfeld |
| 2 | Jerome Seinfeld |
| 3 | Joan Rivers |

**Figure 3.1:** Schema and example data for a student directory application (reproduced from Figure 1.1).

it possible both to limit what portions of database tables a program may access (e.g. only rows with id greater than 10) and to restrict what operations a program may perform on those tables (e.g. reads but not writes). Basing an interface on capabilities is an attractive starting point because capabilities make it easy to limit the authority of a computation. This means that one can understand what database access a program has just by examining what capabilities the program can access.

This chapter presents CapQL, a capability-based database interface where capabilities are used to represent database *views*, a common DBMS abstraction for stored queries. CapQL provides operations for deriving new view capabilities from existing capabilities (e.g. projecting a view) and for fetching or manipulating a view capability's underlying data (e.g. updating the rows in a view). Chapter 4 then shows how ShillDB's capability-safety makes it possible to use CapQL to write secure database-backed applications.

The rest of this chapter is organized as follows. Section 3.1 provides additional background on database views and their applications to access control. Section 3.2 overviews the design goals for CapQL. Section 3.3 presents the operations provided by CapQL. Finally, section 3.4 describes the implementation of CapQL.

## 3.1 Database Views for Access Control

A *view* is an abstraction used in DBMSs that represents a stored query. Views are a type of *virtual table*, meaning views can be queried by database users just like regular tables, but they do not have physical storage backing them. Using the student directory schema presented previously (reproduced in Figure 3.1 for convenience), a view that only contains information about Prof. Seinfeld's

```
SELECT name
FROM jerrys_students
WHERE gpa < 3.0;
```

**(a)** A query over a view before the view definition is expanded.

```
SELECT name
FROM (SELECT name, email, gpa
      FROM students, advising
      WHERE student = id and advisor = "Jerome Seinfeld")
WHERE gpa < 3.0;
```

**(b)** The same query after the view definition is expanded.

**Figure 3.2:** A query over a view is evaluated by expanding the view definitions. The resulting query is unnecessarily complex, but the DBMS can optimize the query.

students can be defined by:

```
CREATE VIEW jerrys_students AS
SELECT name, email, gpa
FROM students, advising
WHERE student = id and advisor = "Jerome Seinfeld";
```

When a user queries a view, the DBMS substitutes the view definition into the query (Figure 3.2).

Most DBMSs allow creating views with a `WITH CHECK OPTION` clause. This option prevents rows from being inserted into the view if they do not satisfy the view's `WHERE` clause and prevents updates that would cause a row in the view to no longer satisfy the `WHERE` clause. In effect, these conditions prevent inserting rows that fall outside of the view or updating rows in the view in a way that causes them to leave the view. When run on the data in Figure 3.1, the following example fails because the update would cause the entry for Mike Birbiglia to leave the `low_gpas` view:

```
CREATE VIEW low_gpas AS
SELECT *
FROM students
```

```
WHERE gpa < 3.0
WITH CHECK OPTION

/* Inflate everyone's grade to an A- ... */
> UPDATE low_gpas
  SET gpa = 3.7;

ERROR: CHECK OPTION failed 'low_gpas'
```

Most DBMSs support using views for *content-based access control* [4]. Content-based access control decides access to data based on the content of the data. Access control policies based on content are natural in a relational database because the data is highly structured. A simple content-based access control policy could be denying a user access to database rows if the id of the row is less than 5. If users cannot access a particular database table except through a view, the view's definition can enforce content-based access by specifying what subset of the table users can access. Views typically can specify what privileges users have on them, similar to the permissions that can be granted on a regular database table (e.g. granting read access but not inserts, deletes, or updates). Using views for content-based access control also enables expressing policies using the same language as queries.

CapQL lifts the notion of views to the language level, where views capabilities are first-class values representing access to a subset of a database. Programmers can invoke operations on view capabilities to derive new views or to access data. CapQL treats all views as though they were created with a `WITH CHECK OPTION` clause.[*]

## 3.2 Design Goals

CapQL aims to meet the following four goals:

1. Queries can only be written in terms of operations on view capabilities (that is, they cannot reference table names directly).

[*] There is no fundamental reason not to provide the option to create a view without this constraint. However, I find inserting a row into a view and not being able to retrieve it to be unintuitive. Requiring that all views have a `CHECK OPTION` constraint also makes it easier to reason about a program's database access by removing a potential parameter.

2. CapQL clearly separates various database operations (e.g. selecting, updating, joining). In particular, this means specifying a view should be distinct from fetching the view's underlying data.

3. View capabilities are first-class values that can be passed around and from which new views can be derived.

4. CapQL is compatible with commodity DBMSs.

Goals 1-3 facilitate reasoning about the authority of database-based applications that use CapQL. Goal 1 makes it possible to limit what tables and rows a program can access by restricting what capabilities the program can access. Goal 2 enables restricting what operations can be performed on a view capability using the same ontology that consumers of the capability use to access the underlying database. Goal 3 makes it possible to pass view capabilities to programs and to restrict existing capabilities (e.g. using a `WHERE` clause). Goal 4 acknowledges that in order for a new interface to be useful in practice, it must support a variety of DBMSs. A recent survey of developers found that MySQL, SQL Server, SQLite, PostgreSQL, and Oracle were each used by at least 15% of respondents [9].

## 3.3 The CapQL Interface

This section presents the CapQL operations for creating and manipulating view capabilities and their correspondence to familiar SQL operations. Note that all operations (except for view creation) take views as arguments and do not make it possible to refer to database tables by name. This design feature of CapQL makes it possible to limit what database tuples a program can view or manipulate by limiting access to capabilities and provides the basis for reasoning about database access in ShillDB programs (see Chapter 4)

A new view of a single database table can be created be supplying the name of the database and the name of the table:

```
(define students (make-view "directory.db" "students"))
```

The resulting view corresponds to selecting all of the data in the given table, so `students` corresponds to the query:

```
SELECT *
FROM students
```

Note, however, that unlike a SQL `SELECT` statement, CapQL does not execute the query corresponding to the view until `fetch` is invoked on the view (see Section 3.3.2). If the underlying DBMS supports users, then `make-view` can also be supplied the user credentials that to use when performing queries on the database.

### 3.3.1 Deriving views from existing views

Four primitive operations are provided for deriving new views from existing views: `where` (which corresponds to selection), `select` (which corresponds to projection)[†], `join`, and `aggregate` (which is like `select` but allows for aggregations).

The behavior of `where`, `select`, and `join` operations maps closely to the behavior of the corresponding keywords in SQL. The `where` operation takes a view and a `WHERE` clause and returns a new view that has been restricted using the given `WHERE` clause. The view defined as:

```
(define w (where students "gpa < 3.0"))
```

corresponds to the query:

```
SELECT *
FROM students
WHERE gpa < 3.0
```

[†] These names may be confusing to programming language specialists familiar with the terms *select* and *project* in the context of relational algebra. However, I believe they are intuitive to SQL users who think of selection in terms of the `WHERE` keyword and projection in terms of the `SELECT` keyword.

Likewise, select takes a view and a list of columns and returns a new view that contains the specified subset of columns from the original view. As in SQL SELECT statements, the columns need not be simple column names but can instead be expressions containing columns. For example:

```
(define s (select students "name, gpa > 3.0"))
```

corresponds to the SQL query:

```
SELECT name, gpa > 3.0
FROM students
```

join takes two views and, optionally, a WHERE clause to restrict the resulting view after the join. Without a WHERE clause, the join will be a full cross-join. To join the students and advising table on student id, one could write:

```
(define students (make-view "directory.db" "students"))
(define advising (make-view "directory.db" "advising"))

(define j (join students advising "id = student"))
```

corresponding to the SQL query:

```
SELECT *
FROM students
JOIN advising
WHERE id = students;
```

Note that supplying a WHERE clause as an argument to join is semantically equivalent to applying a where operation with the same clause after the join; however for the purposes of writing security policies on join operations, it is useful to be able to consider the WHERE clause when determining if the join is allowed (see the discussion of join contracts in Chapter 4).

The `aggregate` operation does not correspond to a single SQL keyword but instead encapsulates features used when performing aggregation queries. The operation takes a view and requires a list of column names which can contain aggregation operations, unlike the argument to `select`. Optionally, `aggregate` takes `GROUP BY` or `HAVING` clauses as keyword arguments.[‡] For example, to calculate the number of students advised by each professor who advises at least five students, one could write:

```
(define agg (aggregate students "advisor, COUNT(*)"
              #:groupby "advisor"
              #:having "COUNT(*) > 5"))
```

corresponding to the SQL query:

```
SELECT advisor, COUNT(*)
FROM students
GROUP BY advisor
HAVING COUNT(*) > 5
```

From an access control perspective, it is useful to distinguish between `select` and `aggregate`. Chapter 4 demonstrates that writing contracts involving the `aggregate` operation allows security policies that only permit a program to see aggregate data.

### 3.3.2 Retrieving data in a view

As mentioned, specifying a view and bringing the data in that view into memory are distinct operations. Given a particular view, the `fetch` operation actually executes the corresponding query and fetches results. Executing the following returns the data in the view represented as a list of rows:

[‡] Restricting a view with a `HAVING` clause is provided as part of the `aggregate` operation rather than as its own operation because in SQL, `HAVING` clauses are only semantically valid when combined with a `GROUP BY` clause.

```
> (fetch
    (select
     (where
      (make-view "directory.db" "students")
      "gpa < 3.0")
     "name, email"))

'(("name", "email")
  ("Mike Birbiglia", "birbigs@college.edu"))
```

### 3.3.3 Modifying view data

CapQL provides `insert`, `update` and `delete` operations that correspond to their SQL counterparts.

The `insert` operation takes a view, a list of column names in the view (with no duplicates), and a list of values for those columns. The operation then inserts the new row into the view. Any columns in the view left off the list must have default values defined in the schema. If the `students` table has the `id` column set to automatically increment and provides a default GPA value, one could write:

```
> (insert students "name, email" '("John Mulaney", "jmulaney@college.edu"))
```

to insert a new student into the table.

The `update` operation takes a view and a list of update statements (such as one would find in the `SET` clause of a SQL update query). Optionally, `update` can be provided a `WHERE` clause, which restricts which rows will be updated. Note that this `WHERE` clause does *not* restrict the values that the updated rows can have, unlike using `where` to restrict the view. The following update will fail analogously to the `CHECK OPTION` example presented in Section 3.1:

```
> (update
    (where students "gpa <= 2.5")
    "gpa = 3.7")
```

```
update: violated view constraint: gpa <= 2.5
```

While the following update will succeed:

```
> (update
    students
    "gpa = 3.7"
    "gpa <= 2.5")
```

The `delete` operation deletes all of the rows in the view. For example, to remove all of the students with poor grades from the table, one could write:

```
> (delete (where students "gpa <= 2.5"))
```

Operations that modify data are not necessarily well-defined for all views. For example, if a view contains a non-simple column (such as `gpa + id`), it may be not be clear what an update to this column would entail for the original view. CapQL does not attempt to provide a complex solution to this *view update problem* (although different solutions have been proposed, such as a new bi-directional query language that allows interpreting view definitions as update policies [5]). Instead, conservative restrictions are used to determine what sort of views are insertable, updatable, and deletable. These restrictions are similar to those found in commercial DBMSs [42, 22].[§]

*Insertability* is a property of a an entire view. A view is considered insertable unless:

- The view references the same base table column multiple times.
- The view is the result of a join.
- The view is missing a column in the base table which does not have a default value.
- The view contains derived columns, such as columns containing aggregation functions, literal values, or expressions over columns.

[§] Some DBMSs, such as MySQL, allow certain updates and insertions on views that are the result of some types of joins. Here, for simplicity, any operations that modify the result of a join are prohibited.

- The view contains a group by or having clause.

*Updatability* is a property of a column in a view, although certain operations (joins or aggregations) can cause all of the columns in a table to no longer be updatable. A column is updatable unless:

- The view is the result of a join.
- The column is a derived column.
- The view is the result of an aggregation.

*Deletability* is a property of an entire view. A view is deletable unless it is the result of a join or an aggregation.

### 3.3.4 Limitations

CapQL does not support all features of SQL, such as different types of joins, union and intersection operations, or nested queries in `WHERE` clauses. It also does not provide functionality for running a sequence of queries in a transaction. These limitations are not fundamental but rather the result of prioritizing the features most important to the larger goal of running database-backed applications with least privilege.

## 3.4 Implementation

I have implemented a prototype of CapQL in Racket [15] on top of Racket's standard database library [8]. View capabilities are implemented as structs that store metadata (such as information about the underlying table's schema) along with an abstract, database-independent version of the query that the view represents. Operations that produce new views from existing views work by manipulating the abstract query representation. This abstract query is only concretized into SQL syntax when an operation is invoked that requires issuing a database query (that is, `fetch`, `update`, delete, or insert). While most of the implementation is database-independent, some SQL syntax is database-specific, and so operations that issue database queries must call out to different functions depending on what DBMS stores the underlying data. CapQL currently supports SQLite3 [36], and supporting other DBMSs would be straightforward.

Executing SQL queries corresponding to CapQL views does not involve DBMS-level views. The fetch and delete operations are easily implemented without DBMS-level views, as CapQL can generate a query corresponding to the fully-expanded view definition without actually creating any intermediate views at the DBMS level.

The update and insert operations do, however, need a mechanism for verifying the CHECK OPTION constraints. These constraints are enforced by installing an insert or update *trigger* [35] on the underlying DBMS table, executing the query, and then removing the trigger. A trigger is a database operation performed in response to a particular event, such as an insert query. The trigger that CapQL installs for insert operations checks if the newly inserted row satisfies the WHERE clause of the view on which the operation was invoked. If the row does not satisfy the WHERE clause, the insert is aborted. Similarly, for an update, the installed trigger checks that each row that would be updated satisfies the WHERE clause of the view. If any row would not satisfy the WHERE clause when updated, the entire update is aborted and no rows are modified. Leveraging existing trigger functionality of DBMSs is preferable to validating updates and insertions in the implementation of CapQL before issuing the query because this would require implementing a SQL interpreter and performing unoptimized DBMS-like functionality in the language runtime. Further, using SQLite's temp trigger functionality [35] (which installs a trigger just for the current database connection), the trigger-based approach works even if multiple applications with different access control policies access the same tables concurrently. Other DBMSs besides SQLite3 have similar trigger mechanisms, and so this approach is portable.

To provide early detection for invalid or malicious operations on a view, CapQL parses and validates any user-provided SQL expressions. The SQL parser is implemented using Racket's standard lex and yacc-style parsing tools [29] and supports a limited but representative set of SQL expressions (boolean and arithmetic expressions over columns, strings, and number literals).

This validation feature is primarily for usability rather than security: Racket's standard database library provides robust checks against SQL injection (such as a user providing `"1; SELECT * FROM secrets"` as a `WHERE` clause), and the underlying DBMS will reject queries that correspond to invalid view operations (for example, referencing a column that has been projected away). However, waiting to report an error back to the user until a `fetch`, `update`, `insert`, or `delete` is executed would make it very difficult to debug the source of the error. This difficulty is exacerbated both by the fact that the error would be from a SQL query generated by CapQL (rather than a query the user directly wrote themselves) and by the typically inscrutable syntax errors provided by DBMSs. Thus, there is a tradeoff between having to re-implement parsing and validation already done by the DBMS and being able to provide timely and informative errors to users. Nonetheless, the merits of providing such detection seem commensurate with the implementation effort.

# 4

# Design and Implementation of ShillDB

ShillDB is a programming language designed to support running database-backed applications with only as much database access as they need to perform their legitimate function. In most programming languages, a program can access any resources that the user running the program can access. Typically, a program inherits the *ambient* authority of the invoking user, which almost always confers far more privilege than the program needs. Further, it is only possible to reason accurately about what resources the program accesses by reading through every line of its source code. By contrast, a ShillDB program's access to database resources is based on *capabilities* rather than the ambient authority of the invoking user. Access to database tables in ShillDB is only possible by invoking operations on CapQL view capabilities. Every ShillDB function also comes with a *contract* which can be used to enforce fine-grained access control policies on view capabilities passed as arguments. Contracts can specify what *privileges* are required on a capability, where privileges represent the authority to invoke a particular CapQL operation. For example, a contract may specify that a view capability passed as an argument only has the +fetch privilege, meaning that the view is read-only. Privileges also support *modifiers* to further refine the privilege, such as restricting a +update privilege to only allow certain rows in the view to be updated. ShillDB contracts thereby serve as *executable documentation* for the program's authority. Through this use of contracts and capabilities, a consumer of a ShillDB program can reason about the authority of a program just by examining the contracts on functions and looking at what capabilities the program is given.

Because it is necessary to use ambient authority to create a starting set of capabilities, ShillDB is divided into two languages: an *ambient* language and a *capability-safe* language. Ambient ShillDB programs can use ambient authority to create the initial view capabilities that are passed to a capability-safe program. These ambient ShillDB programs are intended to be very short so that someone running a ShillDB program can quickly see what ambient authority it uses. To this end, the ambient language is highly restricted: ambient programs are only able to create capabilities for database resources, perform a limited set of actions on those capabilities (for example, using a where clause to restrict a view of a table), and invoke capability-safe functions. The capability-safe language is designed to ensure that a capability-safe program can only access the capabilities it is given as initial arguments and capabilities derived from these arguments. Functions defined in the capability-safe language that are exported for use by ambient programs must have contracts attached that specify the privileges they require on capability arguments.

Figure 4.1 shows an example of how ShillDB uses contracts and capabilities to control access to database tables. The ambient program creates a capability for the students table and passes the capability to a capability-safe program. The interface of capability-safe program then applies the contract (view/c +fetch) to the given capability which declares that the only allowed operation on the view is fetch. The contract serves as a wrapper or *proxy object* that receives operations that the capability-safe program invokes on the capability. If the program calls fetch on the capability, the contract will forward this operation on to the view capability, which in turn will fetch the contents of the view from the DBMS. If, however, the capability-safe program invokes an operation that is prohibited by the contract (e.g. update), the wrapper will reject the operation and signal a contract

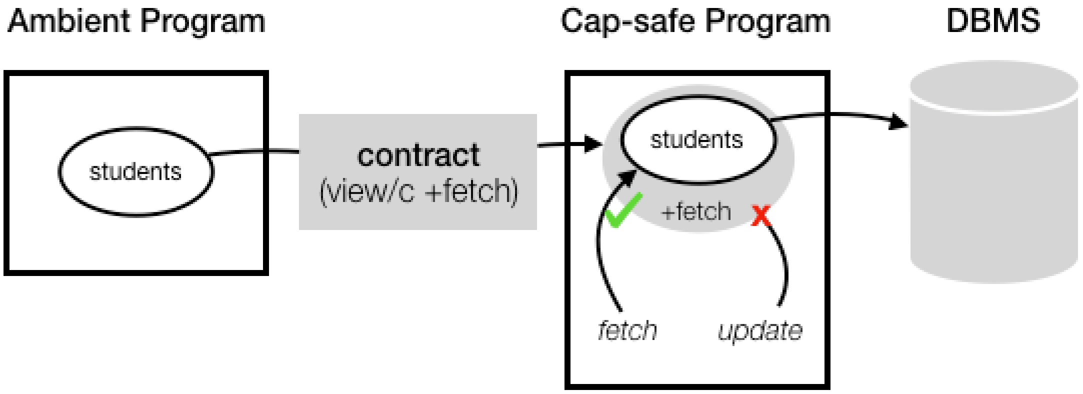


**Figure 4.1:** Contracts and capabilities in ShillDB. Capability-safe programs can only access database tables through capabilities passed in from an ambient program, or capabilities derived from this initial set. This restriction makes it possible to reason about a program's access to database tables just by examining what capabilities the program receives and the contracts on those capabilities. In this example, regardless of the capability-safe program's implementation, the contract guarantees that the only database operation the program can perform is fetching the `students` table.

failure, blaming the capability-safe program. Since the capability-safe program can only access the DBMS through the given capabilities, contracts can mediate all interactions between programs and the DBMS.

The rest of this chapter details how the design and features of ShillDB accomplish the goal of providing database access control tools at the language level. Section 4.1 introduces ShillDB's threat model. Section 4.2 elaborates on capability-safety in ShillDB. Section 4.3 presents ShillDB's contract system and demonstrates how it can be used to write fine-grained security policies at program interfaces. Finally, Section 4.4 describes the implementation of ShillDB.

## 4.1 Threat Model

In ShillDB, programs written in the capability-safe language are treated as though they may be malicious or contain bugs and are therefore untrusted. The database resources that these programs can access are limited by what capabilities they are passed and the contracts on those capabilities. In particular, a ShillDB program has no access to resources for which it does not possess a capability and cannot use capabilities in ways that are disallowed by the capabilities' contracts. ShillDB assumes that contracts on functions are correct insofar as the user executing a ShillDB program wishes to give the program the authority that the contracts specify. Therefore, consumers of ShillDB programs must carefully examine the contracts at the interface of programs before running them. Consumers need not, however, look at the implementation of the program in order to deduce what database access the program has.

ShillDB's trusted computing base includes the ShillDB runtime (and therefore the implementation of the Racket [15] programming language, including its standard database library [8]), and the implementation of any DBMS that a ShillDB program accesses. The operating system and hardware are also trusted. ShillDB therefore does not defend against programs that exploit flaws in the ShillDB implementation, the implementation of Racket, or DBMS implementations. ShillDB does, however, defend against programs that attempt to circumvent its security guarantees through SQL injection.

## 4.2 Limiting Access to Databases Tables

ShillDB programs can only access database tables through view capabilities. Therefore, in order to see or manipulate data in a table, a ShillDB program must have a capability for a view that contains that data. ShillDB provides the CapQL operations on views introduced in Chapter 3; however, the ability to open new views is only available in the ambient language. In the capability-safe language, the only available capabilities are those passed in from an ambient program and capabilities derived from those (for example, by using `where` to apply a more restrictive `WHERE` clause to a view).

*Capability-safety* is enforced by restricting the features available in the capability-safe language. ShillDB does not provide functionality for opening new database connections directly or for reading or writing to files (which could be used, for example, to read or modify SQLite `.db` files). Further, ShillDB does not support serialization or deserialization of view capabilities or allow global mutable state, both of which could be used to store capabilities between function calls. Finally, capability-safe programs can only import definitions from other capability-safe programs. In particular, capability-safe programs cannot import definitions from the ambient language or from students

| *s_id* | *name* | *major* |
|---|---|---|
| 1 | Mike Birbiglia | Biology |
| 2 | Tig Notaro | Computer Science |
| 3 | Patton Oswalt | Computer Science |

courses

| *c_id* | *title* |
|---|---|
| 1 | Standup 101 |
| 2 | The Deal With Airline Food |

ratings

| *course* | *student* | *rating* |
|---|---|---|
| 1 | 1 | 4.0 |
| 1 | 3 | 5.0 |
| 2 | 3 | 1.0 |

**Figure 4.2:** Schema used for a student course ratings application.

Racket libraries that allow direct access to system resources because this could allow a capability-safe program to utilize ambient authority. Although these features limit the expressiveness of the language, they ensure that it is always possible to reason about what capabilities a ShillDB program can access *just* by looking at the initial arguments it is passed.

View capabilities provide a language-level abstraction to represent a subset of the data in a database, either in one table or in multiple tables (as through a join). Thus, by using ambient authority to construct new view capabilities, ambient programs can provide a capability-safe program with access to just the portion of a database that the capability-safe program requires. As an example of the utility of this feature, consider an application that displays the ratings that students have given to different courses. The schema for the data backing this application might look like the one shown in Figure 4.2. In this schema, there is a `students` table that associates student ids with student names and majors, a `courses` table that maps course titles to course ids, and a `ratings` table that stores how a student rated a particular course out of 5. This data is clearly sensitive as it stores individual student opinions of courses, which students expect to be kept private. In particular, the application for viewing the ratings has no legitimate use for the `name` column and therefore should be prohibited from accessing name data.

There are nonetheless many legitimate uses of the data. One may wish to compute the average

```
; cap.rkt
#lang shilldb/cap

(provide avg-rating-by-major)

(define (avg-rating-by-major min-group v-ratings v-students v-courses)
  (println
   (fetch
    (aggregate (join (join v-ratings v-students "student = s_id")
                     v-courses "course = c_id")
               "title, AVG(rating), major"
               #:groupby "title, major"
               #:having (sqlformat "COUNT(course) >= $1" min-group)))))
```

**(a)** A capability-safe program that receives view capabilities and performs CapQL operations on them.

```
; amb.rkt
#lang shilldb/ambient

(require "cap.rkt")

(define ratings (open-view "database.db" "ratings"))
(define students (select
                   (open-view "database.db" "students")
                   "s_id, major"))
(define courses (open-view "database.db" "courses"))

(avg-rating-by-major 10 ratings students courses)
```

**(b)** An ambient program that creates new view capabilities and invokes a capability-safe function. The program restricts the `students` capability so that it does not provide access to the `name` column since the program has no legitimate need for name data.

**Figure 4.3:** An example of using an ambient program to create capabilities and invoke a capability-safe program.

rating that students in different majors assigned courses. This can be done without revealing too much about individual student scores provided that the number of students with a particular major in a class is large enough (if a professor knows that there was only one student in her course who studied chemistry, the professor could easily use the per-major averages to deduce that student's rating of her course). Figure 4.3a shows the implementation of a capability-safe function `avg-rating-by-major` which takes in views for the `ratings`, `students`, and `courses` tables as well as an integer argument, `min-group`. The function then retrieves the average score for all courses grouped by the major of the students who assigned the rating, but filters out results where a major had fewer than `min-group` students represented in a particular class. The `#lang shilldb/cap` annotation on line 2 indicates that this file is a capability-safe ShillDB program. Line 4 exports the function for use by ambient programs. The body of the function consumes three view capabilities and uses them to derive new view capabilities (by using `join` and `aggregate` on lines 9-13) before finally `fetch`ing the desired data and printing it.

Figure 4.3b shows an ambient program that invokes `avg-rating-by-major`. The `#lang shilldb/ambient` annotation indicates that this file is an ambient ShillDB program. Line 4 imports the definition of `avg-rating-by-major` from the capability-safe program shown in Figured 4.3a. Lines 6-10 define view capabilities that will be passed to the `avg-rating-by-major` function. Note that the ambient program is able to create these capabilities using global resource names (that is, the names of database files and tables in the database), whereas the capability-safe program cannot use these names to refer to database tables. If the underlying DBMS supports database users, then user credentials would also be supplied in the ambient code when the capabilities are created. On line 7, the ambient script restricts the `students` view by projecting away the `name` column before passing the view to `avg-rating-by-name` because the function has no legitimate use for student names. Since view capabilities do not provide any operation that can undo this projection, this prevents the capability-safe code from viewing the `name` column. On line 12, the ambient program invokes the imported function with the view capabilities constructed above and passes 10 as the minimum group size.

## 4.3 Contracts on View Capabilities

Capability-safety makes it possible to restrict the database access that a ShillDB program has by controlling what capabilities the program is given, but it does not provide any guarantees on how the given capabilities will be used. Although the informal specification of `avg-rating-by-major` states that it should not reveal average ratings representing fewer than `min-group` students' opinions, the version shown above provides no guarantee that the implementation will only use the given capabilities as desired. In this small example, it is easy enough to read through the database operations and deduce that the function implementation obeys this specification; however, in a real application, the function might make many queries, and the database queries might be intermingled with other code irrelevant to this security concern, making it much more difficult to examine the implementation and decide if the database operations satisfy the intended security policy.

To remedy this, ShillDB allows writing contracts on capabilities to provide fine-grained, runtime-enforceable security specifications for programs. Every function that a ShillDB program exports must be accompanied by a contract which places restriction on what arguments can be passed to the function and how they can be used (this was omitted in Figure 4.3a for simplicity). Consider a simple Racket contract on `avg-rating-by-major`, which might look like:

```
(provide [avg-rating-by-major
          (-> integer? view? view? view? void?)])
```

The **`provide`** form can be used to attach a contract to an exported value. In this case, the contract specifies that the function takes an integer and three view capabilities (for the `ratings` table, the `students` table, and `courses` table, in that order) and returns nothing. While this contract performs basic checks on the arguments and return value of `avg-rating-by-major`, it still does not restrict how the capabilities can be used.

ShillDB provides contract combinators that make it possible to specify a more precise contract on `avg-rating-by-major`. To restrict which operations can be invoked on each view, one could use the following revised contract:

```
(define jfa/c (view/c +join +fetch +aggregate))

(provide [avg-rating-by-major
          (-> integer? jfa/c jfa/c jfa/c void?)])
```

The ShillDB `view/c` contract combinator makes it possible to specify what *privileges* a function requires on a view capability. Each privilege (such as `+join` or `+fetch`) represents the right to invoke the corresponding CapQL operation on a capability. This revised contract will protect against certain obviously buggy or malicious implementations of `avg-rating-by-major`, such as one that attempts to update or delete rows in any of the tables. The contract is not, however, sufficient to enforce finer-grained security policies, such as requiring that any rating information revealed should reflect at least `min-group` number of student opinions.

Two more of ShillDB's contract features are needed to enforce the desired access control policy for this function. The first is *privilege modifiers*, which can be used to further refine what operations a privilege permits. Figure 4.4 shows all of the supported modifiers for each ShillDB privilege. As an example, the `+aggregate` privilege supports a `#:with` modifier which can be used to specify the contract that a view should derive after an aggregation operation. This modifier is useful for the student directory application because one can specify that the view containing ratings only derives a `+fetch` privilege after an appropriate aggregation is applied.

The second necessary feature is *join groups*, which provide a way to express constraints on how particular views can be joined. Note that while most operations on view capabilities only operate on a single capability, `join` is inherently *binary*. In many cases, it is natural to think about how a particular set of views can be joined together, rather than trying to write appropriate modifiers for `+join` privileges on individual views. To this end, ShillDB provides the `->/join` and `->i/join` contract combinators for writing *join group* and *dependent join group* constraints on function arguments, respectively. A join group confers the privilege that any views in the group to be joined together. The `#:pre` modifier for unary join privileges is replaced by the ability to add particular views to the group using the `#:groups` annotation in the contract. A join group allows for `#:post` and `#:with` modifiers similar to the unary join privilege modifiers. A dependent join group differs from a regular

**Privileges and Modifiers**

| *Privileges* | *Modifier* | *Description* | *Example* |
|---|---|---|---|
| `+fetch`<br>`+update`<br>`+delete`<br>`+insert` | `#:restrict` | Provides a restricted window into the view for a particular operation based on the given view to view function. | `#:restrict (λ (v) (where v “id < 10”))` |
| `+aggregate` | `#:having` | Filters out any groups in the resulting view that do not satisfy the given `HAVING` clause. | `#:having “COUNT(*) > 10”` |
|  | `#:aggrs` | Rejects any aggregation query that contains an aggregation function other than those listed. | `#:aggrs “MIN, MAX”` |
|  | `#:with` | Specifies what contract the view should derive after an aggregation. | `#:with (view/c +fetch)` |
| `+join` | `#:pre` | Rejects any joins that do not satisfy the given predicate over tables and join condition. | `#:pre valid-foreign-key?` |
|  | `#:post` | Applies the given view to view function to the result of joins. | `#:post (λ (v) (select v “id”))` |
|  | `#:with` | Specifies what contract the view should derive after a join. | `#:with (view/c +fetch)` |
| `+where`<br>`+select` | ——— | ——— | — — — |

**Figure 4.4:** Privileges and modifiers in ShillDB. Modifiers can be used to refine what operations a particular privilege permits. `valid-foreign-key?` in the example used for `#:pre` is a function that takes two views and a where clause and returns true just when either of the tables corresponding to the views has a foreign key field for the other table and the where clause represents an equijoin on that key.

```
1  (->i/join
2   ([X (min-group)
3       #:post (lambda (v) (select v "course, rating, major"))
4       #:with (view/c +join +select
5                      [+aggregate
6                       #:having (sqlformat "COUNT(*) > $1" min-group)
7                       #:with (view/c +fetch)])])
8   ([min-group integer?])
9   [(view/c +join) #:groups X]      ; v-ratings
10  [(view/c +join) #:groups X]      ; v-students
11  (view/c +join +fetch +aggregate) ; v-courses
12  void?)
```

**Figure 4.5:** A contract for `avg-rating-by-major` (the **`provide`** form is omitted for brevity). Using a contract to express security policies lets one specify policies in a declarative style and guarantees that ShillDB will enforce the policy regardless of the implementation of `avg-rating-by-major`. The contract also allows a consumer of the program to read and understand the security specification without looking at the implementation and makes it easy to modify the specification and implementation separately.

join group in that the definition of the group can depend on the value of a function argument.*

Figure 4.5 combines all of the features of ShillDB contracts together to create the final version of the contract for `avg-rating-by-major`. Lines 2-7 create a dependent join group bound to the identifier `X` that depends on the value of the `min-group` argument. The join group requires that after any join involving views in the group, the resulting view should be projected so the potentially sensitive student ids are hidden (line 3). The join group also specifies the contract that the resulting view should have (lines 4-7). In particular, this derived contract specifies that it is only possible to fetch the resulting view after first aggregating the view (line 7) and that the aggregation must filter out any groups that do not satisfy the group size requirement (line 6). The rest of the contract specifies contracts on each of the function arguments: `min-group` must be an integer (line 8), `v-`

*Note that the functionality of join groups cannot be accomplished simply by writing `#:pre` predicates that check table names for two reasons. First, while table names uniquely identify tables in a database, they do not uniquely identify view capabilities. A function may reasonably consume multiple capabilities that represent different views of the same table and have different associated privileges. Second, a ShillDB function may be designed generically such that one of its arguments is a view that could represent different tables. Thus, to properly allow for constraints on how views can be joined, it is necessary to be able to write contracts that constrain which *capability* arguments can be joined together.

ratings and v-students are views in the join group defined earlier (lines 9-10), v-courses is a view with +join, +fetch, +aggregate privileges (line 11), and the function returns nothing (line 12).

This contract enforces two important security policies in avg-rating-by-major. First, it is not possible to view course rating data unless it is the result of aggregating at least min-group students' opinions. This policy is enforced by making sure the only way to derive the +fetch privilege on a view containing rating data is to aggregate the result of joining the ratings and students tables together, with the restriction that this aggregation will filter out any groups that are too small. Second, the policy prohibits seeing course ratings associated with student ids. This is the result of the post condition of the join group constraint which projects away the student and s_id columns after students and ratings are joined together. Since both views prohibit fetching their contents until after they have been joined (and then aggregated), it is also not possible to fetch both views individually and perform the join manually using the results. The application legitimately needs access to student ids to join the ratings and students tables, but it is desirable to prohibit viewing ids and course ratings together in case student ids are generally accessible to professors (perhaps in a student directory application).

View contracts function in two different ways. First, they might prohibit an operation outright. Second, they may allow an operation but modify its effect. Simple privileges are an example of the first sort of contract: calling fetch on a capability without the fetch privilege will result in a contract failure, and program execution will stop. Similarly, attempting to join a view with another view that is not in the same join group can cause a contract failure. However, consider the following contract:

```
(view/c +fetch
        [+delete #:restrict (lambda (v) (where v "dept = 10 OR dept = 11"))])
```

This contract provides different views of the underlying table depending on which operation is invoked on the capability. In particular, the contract states that all the contents of the view can be fetched, but the `#:restrict` modifier restricts the delete privilege so that only rows where the `dept` column is 10 or 11 can be deleted. This restriction on deletes is enforced through *dynamic query rewriting*: any delete operations invoked on this view are rewritten to prevent queries from violating the security policy. In this case, rewriting queries requires adding an extra term to the `WHERE` clause of any delete queries to enforce the restriction. This rewriting approach requires a much simpler implementation than statically validating queries and less runtime overhead than dynamically checking query results. The same technique is used to enforce `#:restrict` and `#:having` modifiers on other operations.

Internally, SHILLDB contracts work by wrapping capabilities passed into a function in *proxy objects*. The body of a contracted function does not receive the capabilities directly but instead receives the proxies. This allows the SHILLDB runtime to intercept calls to operations on capabilities and choose to allow, deny, or modify the operation before passing it on to the capability. If a proxy intercepts an operation that is prohibited by contract, it will signal a contract failure and abort execution. The contract violation error also includes blame information about which part of the program is at fault for the policy violation to aid in debugging.

## 4.4 Implementation

I have implemented a prototype of SHILLDB in Racket [15] using Racket's macro system and tools for creating languages [41]. Racket's macro system allows writing functions from syntax objects to other syntax objects (and is thus a form of source-to-source compilation). SHILLDB syntax is defined using macros that expand into Racket code, which can then be run using the standard Racket runtime. SHILLDB contract combinators like `->i/join` present a syntactic interface for users to write contracts but are compiled into a contract written using Racket's contract system. Implementing SHILLDB in this way allows using Racket's contract implementation and also makes it easy to reuse Racket features that do not compromise security (for example, functions, `let` statements, or Racket's standard math library).

```
(define (join v1 v2 [join-on ""])
  (((compose (view-join-constraint v1)
             (view-join-constraint v2))
    join-impl)
   v1
   v2
   join-on))
```

**Figure 4.6:** The implementation of the `join` operation on view capabilities. `join` takes two view arguments and (optionally) a where clause to apply after the join. `view-join-constraint` is an operation on views (that is not exposed to ShillDB users) that retrieves the join constraint stored for a particular view. `join-impl` (not shown) is the actual, internal implementation of the join operation and has the same signature as `join`. The implementation of `join` applies the join constraint contract projections to the internal join implementation and then passes the user-supplied arguments to the contracted result.

### 4.4.1 Capabilities and Contracts

ShillDB uses CapQL to provide a capability-based interface for accessing database resources. Each view capability is an object-like value that encapsulates information used to access a database, such as the database and table name. Operations on capabilities either produce a new capability (e.g. `where`) or call a corresponding operation on the actual database (e.g. `update`). Each operation has a corresponding privilege that must be present in order for the operation to be invoked on the capability.

ShillDB contracts are implemented using Racket's contract facilities and create proxy objects around capabilities, allowing contracts to interpose on operations and check privileges. Contracts also allow for privilege modifiers which can restrict or modify the arguments to an operation. For example, the `#:restrict` modifier on a `fetch`, `update`, `delete`, or `insert` privilege modifies an operation so that it applies a more restrictive `WHERE` clause to the view before the original operation is invoked on the view capability. The proxy objects are implemented as Racket struct impersonators [39] which allow redirecting or modifying operations on a struct, making it possible to support privileges and modifiers.

Because joins are binary, the `join` operation is not implemented as an operation on individual capabilities. Instead, `join` is a regular function that consumes two view capabilities. Individual view capabilities can store a contract projection [14] that the capability will apply to the `join` function. A contract projection is a function that defines a contract's behavior; effectively, it is a function that takes a value and returns the value after applying the contract to it. Within the implementation of `join`, each of the views' *join constraint* projections is applied to the internal implementation of the join operation before invoking the contracted join implementation on the capabilities (Figure 4.6). View proxies also store blame information so that violations of constraints on joins blame the correct party based on which view's contract was broken.

Join groups are thus implemented by storing a join constraint in a proxy object around each of the views in the group. It is not possible to implement this functionality using Racket's standard dependent contract combinator, **`->i`**, as this would result in circularly dependent contracts. Specifically, if two views are in a join group together, then the contract on each view must be defined in terms of the other view. This makes it impossible to determine an appropriate order to evaluate the contracts, since correct blame requires that a dependent contract be evaluated only after the depended-on value's contract has already been applied to that value [10]. Instead, the implementation of join groups uses a trick where each view in the group stores a mutable list of the views with which it can be joined. As a join group contract is applied to the arguments of a function, each view in the group is added to the lists of the other views in the group. Within the body of the function, the lists will have been populated and the join constraints will work as desired. The one caveat is that applying the contract to the arguments must not result in performing `join` on any of the arguments, as the join constraints could be in an inconsistent state while the contract is being evaluated.

The ShillDB `->/join` and `->i/join` contract combinators are macros that hide the details of the implementation of join groups from users. Users write relatively simple contracts which expand out to a much longer and more complex Racket contract. However, `->/join` and `->i/join` do not provide support for the full range of possibilities that Racket's function contract combinators provide (such as optional arguments or dependencies between function arguments). In the future, I plan to devise a better implementation for join group constraints that is still easy to use but can be more easily composed with Racket's existing function contract combinators, thus avoiding the need to reimplement complicated functionality provided by built-in combinators like `->i` and `->*`.

### 4.4.2 Language Restrictions

In order to provide capability-safety guarantees, it is necessary for ShillDB to provide a restricted subset of the functionality available in Racket.

#### Capability-safe Language

The capability-safe language does not allow access to certain Racket standard libraries that could allow accessing databases using ambient authority, such as the system library or the database library. Mutable global variables are also not allowed, as this could allow a function to store capabilities in global state between calls. View capabilities cannot be serialized or deserialized, as this could allow storing serialized capabilities in database tables. Finally, ShillDB capability-safe programs can only import definitions from other capability-safe programs to prevent gaining access to functions written in other languages that allow the use of ambient authority (such as Racket or the ambient language).

#### Ambient Language

The ambient language is even further restricted. Ambient programs are intended be short and simple in order to facilitate easy reasoning about what ambient authority a program uses. Accordingly, the language provides a minimal set of features. The ambient language can only import definitions from capability-safe ShillDB programs, create base values like numbers and strings, define immutable values, invoke functions, and perform basic operations on view capabilities (such as restricting views with a `WHERE` clause). The one feature the ambient language has that the capability-safe language does not is the ability to use ambient authority to create new view capabilities by referencing global database and table names.

### 4.4.3 Language Interoperability

The only way to receive the security guarantees provided by ShillDB is to write programs using the language. In the worst case, this means rewriting entire applications in ShillDB, which may be arduous (due to the lack of features like mutable state) or impossible (due to the inability to read or write to files). An ideal solution to this problem would be to add functionality to ShillDB that allows extending ShillDB security guarantees to arbitrary executables that a ShillDB program invokes. However, being able to intercept database queries issued by arbitrary executables would require significant future work and likely necessitate specialized database drivers or DBMSs.

Lacking this functionality, an alternative is writing sensitive portions of an application in ShillDB and writing the rest of the program in a more feature-rich language in the *Racket ecosystem*, such as Racket or Typed Racket. Racket's implementation allows composing components written in different languages within the Racket ecosystem [13]. This feature makes it possible for ambient ShillDB programs to call functions written in the capability-safe language. By the same mechanism, a Racket program could import a function definition from an ambient program and invoke that function. While the entire program would not have ShillDB's security guarantees, capability-safe functions still retain their security guarantees. This technique also allows cleanly separating sensitive, database-accessing components from the rest of the program. Thus, while the whole program would still have ambient authority and would not run with least privilege, the interfaces of any ShillDB components would still clearly specify and constrain the privileges that the components require.

For this reason, I have extended the ambient language with the ability to define and export functions for use by other languages in the Racket ecosystem. Components that access database resources can be written in ShillDB to enforce access control policies while the rest of the program need not deal with the restrictions imposed by ShillDB. Designing programs in this way has the added benefit of encouraging authors to separate components of the program that can access databases from those that cannot. For this approach to be effective, it is still necessary for consumers of a program to inspect the contracts on ShillDB components and make sure that components written in

Racket do not use ambient authority to access databases.

# 5

# ShillDB in Action

To evaluate the usability and performance of ShillDB, I have used it to implement a library reservation system. I have also implemented benchmarks for CapQL operations to better understand the performance characteristics of individual operations. My evaluation indicates that ShillDB can be used to enforce fine-grained database security policies in realistic, multi-user applications with reasonable performance overheard and suggests directions for future performance optimizations.

Section 5.1 describes the library reservation case study. Section 5.2 presents and analyzes performance results for ShillDB and CapQL.

## 5.1 Case Study

I have used ShillDB to implement a secure library reservation backend. The backend provides endpoints for common features on a library website such as looking up authors and books, checking a user's current reservations, and adding or removing reservations. Unlike a typical web application, the backend also has a detailed security interface for each endpoint. For example, the endpoint that allows users to look up information about books has a contract specifying that the function can access the aggregate number of current reservations for a book but not the details of who holds those reservations. The specifications on endpoints that allow users to view or modify their reservations have contracts restricting that only the logged-in user's information can be viewed/modified.

Some application logic in the library reservation system cannot naturally be expressed as ShillDB contracts. In particular, ShillDB lacks constructs for writing contracts that determine whether an operation is allowed based on the result of another database query. For example, a desirable check might be to prevent a user from reserving the same book multiple times or to limit the total number of books a user can reserve. There is currently no way in ShillDB to write an insert contract to enforce these checks, so they must be written as normal application code. Extending ShillDB's contract system to allow contracts with arbitrary predicates over the arguments to an operation would allow enforcing both of these policies (e.g. one could enforce no duplicate reservations by writing a predicate as part of an `+insert` privilege that looked up and validated the logged-in user's reservations). I believe that this extension would be straightforward to implement and would enable pushing even more of the application's database policies to the program's interface.

The server is written in a combination of Racket and ShillDB code (see Section 4.4.3 for a discussion of language interoperability). The main loop of the server is implemented in Racket (to take advantage of libraries for writing web servers in Racket which have not been vetted as safe to use in ShillDB). The backend functionality that interacts with the database is implemented in capability-safe ShillDB code. A short ambient ShillDB program provides an interface between the Racket and ShillDB code and creates the necessary capabilities for use by the capability-safe implementations of the server endpoints. When the server receives a request, it calls the corresponding ShillDB function to handle the request.

The implementation of the backend server required 35 lines of Racket code and 60 lines of capability-safe ShillDB code (of which 14 lines specify contracts). The ambient program is 20 lines long. Note that this small prototype does not handle user authentication or session management. For simplicity, the endpoints provided by the library server take a user id as an argument. In a real application, the application would determine the id to pass to the capability-safe code based on the id of the currently logged-in user. However, implementing proper user authentication is orthogonal to ShillDB's goal of enforcing fine-grained database security specifications.

## 5.2 Performance Analysis

The prototype implementations of ShillDB and CapQL focus on providing security guarantees and have not yet been optimized for performance. Nonetheless, I used the library reservation system as a performance benchmark to verify that the performance overhead of using ShillDB is not outlandish.

The benchmark considers the performance of the library reservation system using four different implementations. As a baseline, I implemented the reservation system using Racket's standard database library. To examine the overhead of using CapQL instead of the standard database interface, I implemented two different versions of the server in Racket using CapQL. The first uses a modified CapQL implementation that does not install database triggers to enforce view constraints for updates or inserts, while the second uses the CapQL prototype implemented as described in Chapter 3. The final configuration replaces most of the Racket implementation with ShillDB code complete with contracts specifying security policies (and corresponds to the implementation described in the previous section).

For each configuration, I considered three different sorts of request workloads for the server. The first workload consists of a series of 1,500 requests that require both reading from and writing to the database (e.g. looking up books, adding new reservations, deleting existing reservations). The second workload consists of 750 requests that require only database reads (e.g. looking up books and reservations). The final workload consists of 2,000 requests that only require insert options (i.e. reserving books without first looking up information about the books).

I ran each configuration for each workload 50 times and computed the mean time to complete all of the requests sequentially, along with 95% confidence intervals. The performance measurements

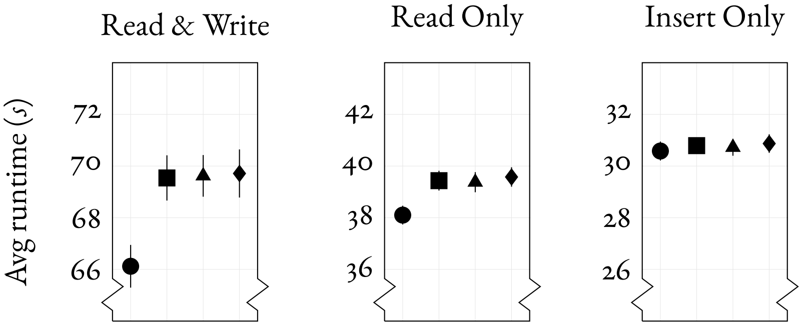


**Figure 5.1:** Average time required to run library reservation system workloads for the baseline (○), CapQL with no triggers (□), CapQL with triggers (△), and ShillDB implementation (◇). 95% confidence intervals are indicated by vertical bars (which may be obscured by plotting symbols when intervals are small). Note that y-axes begin at different values to emphasize differences between configurations for each workload, but the scale is consistent between plots.

were conducted on a four core, 2.7 GHz i7 machine with 8GB of RAM running macOS 10.12.6. Figure 5.1 displays the result.

First, observe that the slowdown for the ShillDB implementations is small for all three workloads: the largest slowdown was 5.43% in the read & write workload. Second, note that the biggest difference in performance comes from going from the baseline to the CapQL implementation. Adding in database triggers to enforce view constraints and adding in ShillDB contracts both result in negligible slowdowns compared to the CapQL implementation. Finally, note that ShillDB and CapQL have comparatively large slowdowns in the read & write and read-only workloads (5.43% and 3.87% respectively) compared to the insert-only workload (0.95%). The following section examines the performance characteristics of different CapQL operations to help better explain this finding. Overall, these results suggest that future performance improvements ought to focus on the implementation of CapQL, with an emphasis on operations likes `fetch`, `where`, and `delete`.

### 5.2.1 CapQL Benchmarks

To better understand the overhead of database operations due to the implementation of CapQL, I evaluated small benchmarks for different database operations using Racket's standard database library and CapQL operations. All of the benchmarks used a simple schema consisting of one table with two integer columns. The where benchmark performs a selection and a fetch for a particular query selectivity.* The delete benchmark deletes a particular portion of the rows in the table. The update benchmark performs a simple arithmetic update on a given portion of the rows in the table. The CapQL implementation of the update benchmark restricts the view before updating using a simple where clause to measure the overhead of installing and executing the database trigger for the view. The insert benchmark inserts ten rows into the table. As in the update benchmark, the CapQL implementation restricts the view before inserting. For both the update and insert benchmarks, there is an accompanying version that uses a modified CapQL implementation that does not install database triggers.

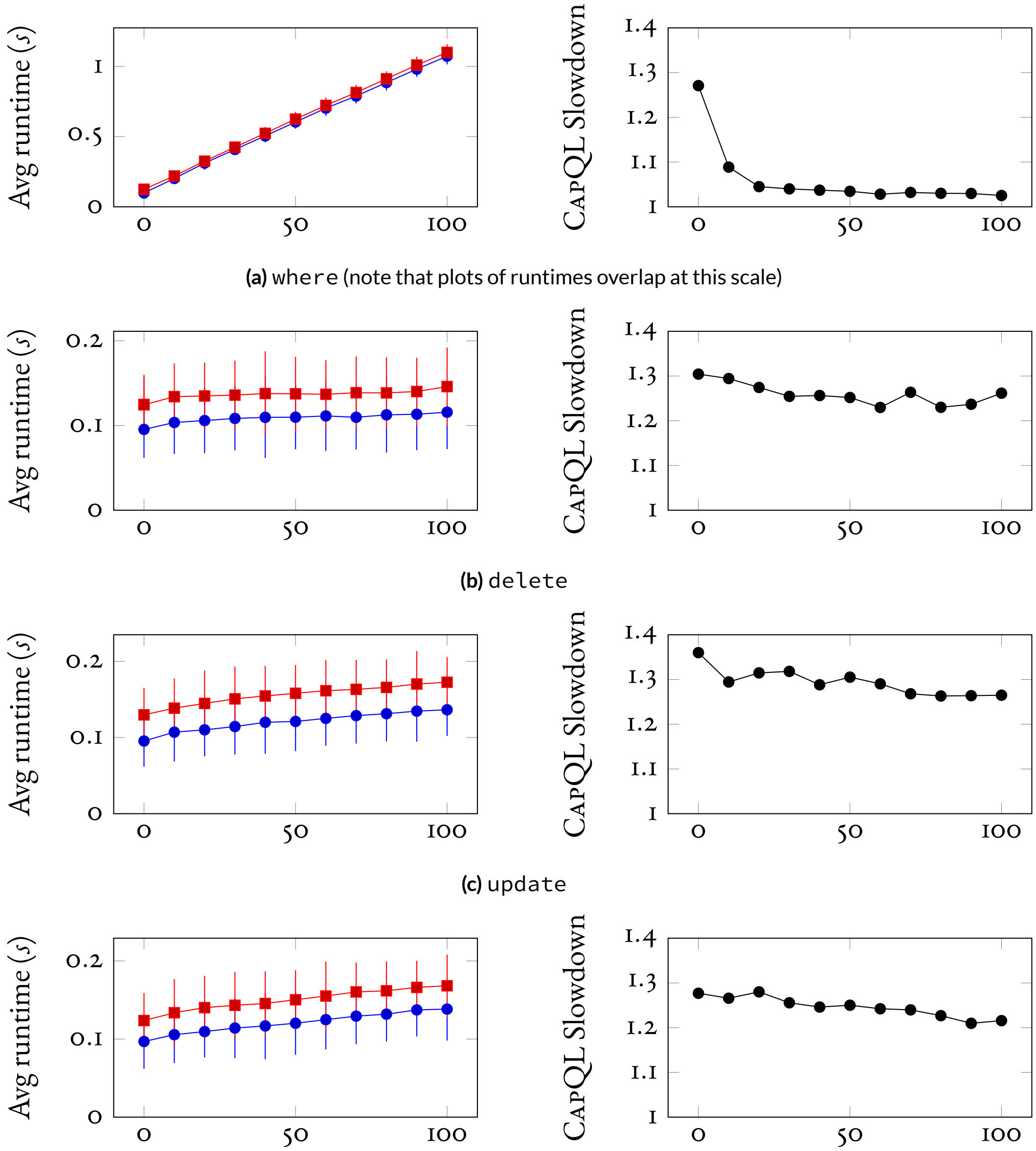
I timed each of the benchmarks 100 times against a table with 50,000 rows for a variety of selectivity values between 0% and 100% (except for the insert benchmarks for which the notion of selectivity does not make sense). Figure 5.2 shows the mean execution times with 95% confidence intervals (left-hand column) and the mean slowdown for the CapQL implementation compared to the baseline (right-hand column). The mean slowdown for the insert benchmark (not shown in the figure) was 1.08× with triggers and 1.02× without triggers.

First, note that for read operations, the slowdown due to CapQL is negligible for large queries: the time required to execute the query and bring the results into memory dwarfs the overhead. Second, observe that, while the slowdown trends downwards with increasing query size for update and delete operations, the overhead is still significant for large queries. This is not due to the overhead of checking triggers in updates because the trend is consistent for updates, for deletes (which do not install triggers), and for the modified update benchmark with no triggers. Instead, the trend must be due to the fact that the cost of updates and deletes at the DBMS level in this case does not increase as dramatically with increased selectivity as the cost of fetching data, and thus the overhead of CapQL remains significant even for large queries. Finally, note that these results are consistent with the performance results from the library case study. Low selectivity deletes and fetches are the worst case for CapQL compared to the standard database library, whereas the overhead of insertions

*Note that contrary to the meaning of the word *selective*, selectivity refers to the portion of rows that a query affects. A query with high selectivity thus affects a large portion of rows.

is small. This corroborates the result that the read & write workload (which consists primarily of low-selectivity deletes and fetches with some inserts) and the read workload (low-selectivity fetches only) performed much worse than the insert-only workload.

Finally, to understand why certain CapQL operations have such significant overhead compared Racket's standard database interface, I used Racket's statistical profiler [30] to profile one of the CapQL microbenchmarks. I ran the `where` benchmark with 0% query selectivity 10,000 times, and the profiler collected samples to estimate the execution costs for different functions called during the benchmark. Since the profiler estimates execution time using random samples, the results it provides are not a perfectly accurate measure of execution time. Nonetheless, the results suggest that a significant portion of running time (about 1/3) was spent parsing and validating the `WHERE` clause argument supplied to the `where` operation. This helps explain the results of the microbenchmarks, as `where`, `update`, and `delete` all take SQL expressions as arguments which they parse and validate, while `insert` does not have any SQL expression arguments. Further, this suggests that future performance optimizations ought to focus on speeding up validation of SQL arguments or pushing more of this work to the DBMS.



**(a)** `where` (note that plots of runtimes overlap at this scale)

**(b)** `delete`

**(c)** `update`

**(d)** `update` with no triggers installed

**Figure 5.2:** Results for each CapQL benchmark. The left-hand column shows average runtime vs. query selectivity % for the baseline (○) and CapQL (□) configurations. 95% confidence intervals are indicated by vertical bars. The right-hand column plots the slowdown of the CapQL implementation compared to the baseline vs. query selectivity %.

# 6

# Related Work

While there is a rich line of research in access control mechanisms at the DBMS level (see Bertino et al. [4] for a comprehensive overview), it is most useful to consider this thesis in the context of other work that uses *language-based* security techniques. To this end, this chapter considers three categories of related work: approaches to supporting programming with least privilege, various uses of software contracts for application security, and other techniques for enforcing database security policies at the language level.

## 6.1 Programming Language Support for the POLP

Shill is a secure shell scripting language that uses capabilities and contract to limit access to file system and network resources. ShillDB uses the same techniques as Shill [27] to support running applications with least privilege and follows a similar design and implementation. Shill is able to transitively extend its authority guarantees to arbitrary executables using capability-based sandboxes to enforce contracts, which ShillDB does not support. As implemented, Shill sandboxes require installing a special kernel module which is only implemented for FreeBSD. Adding sandboxes to ShillDB would likely mean that language users would have to install specialized database drivers or modified DBMSs, increasing the setup required to use the language (currently, anyone with a Racket installation can download and use ShillDB).

Other systems have also used language-level capabilities to support the POLP. The E programming language [23] is an *object capability language*, where object references are treated as capabilities to invoke operations on that object. Passing an object reference provides unattenuated access to the underlying object, but it is possible to protect sensitive behaviors of objects by wrapping them in proxy objects. Both ShillDB and Shill use contracts to provide a convenient way to express and enforce these sorts of *access abstractions* at component interfaces. CapDesk [38] provides support for launching applications written in E with limited authority. CapDesk allows applications to negotiate more authority from the user after they are launched, which ShillDB does not support. CapDesk, however, does not have a scripting interface for writing security policies.

Melicher et. al [20] propose a module system which treats modules as first-class capabilities. A module can only access another module if it possesses a capability for that module, and capabilities for sensitive *resource modules* can only be obtained as arguments to an ML-style functor [19]. Melicher et al. assume that a system is divided into a trusted code base and untrusted extensions (whereas all ShillDB components are considered untrusted). They then use their module system to reason about the authority that untrusted modules have to access system resources.

Many research efforts have sought to limit mainstream programming languages to capability-safe subsets in order to better support reasoning about and limiting the authority of programs. These approaches typically involve limiting the API of the original language and restricting access to ambient authority. Examples include Joe-E [21] (a subset of Java), Emily [37] (a subset of OCaml), and Google's Caja [24] (a subset of JavaScript). Unlike these capability-safe languages, ShillDB and Shill leverage contracts to enforce access policies on capabilities at function interfaces and aim to limit a program's authority to access specific resources (database tables and file system resources, respectively).

## 6.2 Software Contracts for Security

Prior work has also used contracts to enforce access control policies. Heidegger et al. [16] introduce access permissions contracts to restrict the fields for which a method has read and write permissions on a particular object. Dimoulas et al. [11] use contracts to control how capabilities can flow between components in object-capability languages. Shill [27] uses contracts to limit the ways that file system and network capabilities can be used. ShillDB uses contracts in much the same way, but provides contract features designed specifically for database access control policies.

Each of these systems, including ShillDB, is specialized for enforcing a particular type of access control policy. Other contract systems allow enforcing more general policies. Moore et al. [26] use *authority environments* to manage rights within an execution context and *authorization contracts* to limit authority environments. Disney et al. [12] introduce temporal higher-order contracts that can enforce temporal properties that are common in imperative programs (for example, requiring functions to be called in a specific order).

## 6.3 Language-based Database Security

The approach to language-level database security most closely related to ShillDB's is Caires et al.'s [6] proposal for a refinement type system that can statically check that programs adhere to database access control policies. Data models are defined in an object-relational mapping (ORM) style and security policies can be specified for fields of data objects. In order for an access to be allowed, the necessary conditions for that access must be deducible from the current *knowledge*. For example, the type system could prevent a query that accesses a photo unless the program first checks to ensure that the photo belongs to a friend of the current user. In effect, queries that return policy-violating results are statically rejected. By contrast, ShillDB dynamically rewrites user queries to filter out results that would violate access policies. Caires et al.'s type system also requires programmers to provide knowledge annotations on functions to facilitate type checking. In some cases, the burden of providing annotations (and the limited expressiveness of static types compared to dynamic contracts) may be worthwhile to avoid the runtime overhead of contract checks or to gain static security assurances. Attaching security policies to fields in an ORM definition provides a promising future direction to improve the usability of ShillDB.

Other approaches have considered the problem of *information flow* instead of access control. Information flow is able to address indirect or implicit data flows (for example, branching based on a piece of sensitive information), which access control does not consider. SeLINQ [34] and UrFlow [7] use static types to reject queries that violate information flow policies. Jacqueline [43] performs dynamic query rewriting to enforce information flow policies (like ShillDB does for access policies). Jacqueline uses a *policy-agnostic* approach in which information flow policies are factored out from the rest of the application code and then managed by the framework. This is similar in philosophy to ShillDB, in which access control policies are pushed to function interfaces and then enforced by the language runtime.

# 7

# Conclusion and Research Outlook

ShillDB provides language-based support for running database-backed applications with least privilege and enforcing fine-grained database access control policies. ShillDB is capability-safe and ensures that programs can only access database resources based on the view capabilities they are given, regardless of what user runs the program or what database credentials the program may know. Finally, ShillDB provides contract combinators tailored for writing fine-grained specifications for how database views can be used, and the ShillDB runtime enforces these contracts. This allows pushing access control policies to program interfaces, making it easy to inspect or modify the policy without looking at the program's implementation.

There are at least three possible directions for future work based on this thesis. First, many possible improvements exist for the prototype implementation of ShillDB: in particular, supporting more DBMSs, making certain ShillDB contract combinators easier to compose with Racket's contract system (see Section 4.4), or extending the expressiveness of the contract system (for example, to address the limitations discussed in Section 5.1). Second, extending the security guarantees of ShillDB to arbitrary executables would allow users to take advantage of ShillDB's security features without rewriting their applications. Adding this functionality to ShillDB would likely require custom database drivers that route requests to a database proxy or else modifications to existing DBMSs. Finally, it is worth considering if one could create a more general framework for running applications with least privilege using contracts and capabilities. Such a framework might allow Shill's [27] filesystem and network capabilities as well as ShillDB's view capabilities to exist as plugins in a larger language, reducing the design and implementation efforts needed to support other interfaces.

# References

[1] Facebook photo leak flaw raises security concerns. http://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns. Accessed: 2017-11-28.

[2] Idris language reference. http://docs.idris-lang.org/en/latest/reference/. Accessed: 2017-12-24.

[3] Owasp top ten project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Accessed: 2017-11-25.

[4] Bertino, E., Ghinita, G., and Kamra, A. (2011). Access control for databases: Concepts and systems. *Foundations and Trends in Databases*, 3(1–2):1–148.

[5] Bohannon, A., Pierce, B. C., and Vaughan, J. A. (2006). Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347. ACM.

[6] Caires, L., Pérez, J. A., Seco, J. C., Vieira, H. T., and Ferrão, L. (2011). Type-based access control in data-centric systems. In *ESOP*, pages 136–155. Springer.

[7] Chlipala, A. and Impredicative, L. (2010). Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, pages 105–118.

[8] DB: Database Connectivity. https://docs.racket-lang.org/db/. Accessed: 2017-12-31.

[9] Developer Survey Results 2017. https://insights.stackoverflow.com/survey/2017. Accessed: 2017-12-31.

[10] Dimoulas, C., Findler, R. B., Flanagan, C., and Felleisen, M. (2011). Correct blame for contracts: no more scapegoating. In *ACM SIGPLAN Notices*, volume 46, pages 215–226. ACM.

[11] Dimoulas, C., Moore, S., Askarov, A., and Chong, S. (2014). Declarative policies for capability control. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 3–17. IEEE.

[12] Disney, T., Flanagan, C., and McCarthy, J. (2011). Temporal higher-order contracts. In *ACM SIGPLAN Notices*, volume 46, pages 176–188. ACM.

[13] Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., and Tobin-Hochstadt, S. (2015). The racket manifesto. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[14] Findler, R. and Blume, M. (2006). Contracts as pairs of projections. *Functional and Logic Programming*, pages 226–241.

[15] Flatt, M. and PLT (2010). Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc. `https://racket-lang.org/tr1/`.

[16] Heidegger, P., Bieniusa, A., and Thiemann, P. (2012). Access permission contracts for scripting languages. *ACM SIGPLAN Notices*, 47(1):111–122.

[17] INRIA. Coq. `https://coq.infria.fr`. Accessed: 2017-05-04.

[18] Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer.

[19] MacQueen, D. (1984). Modules for standard ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, New York, NY, USA. ACM.

[20] Melicher, D., Shi, Y., Potanin, A., and Aldrich, J. (2017). A capability-based module system for authority control. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Mettler, A., Wagner, D., and Close, T. (2010). Joe-e: A security-oriented subset of java. In *NDSS*, volume 10, pages 357–374.

[22] Microsoft. Modify data through a view. `https://docs.microsoft.com/en-us/sql/relational-databases/views/modify-data-through-a-view`. Accessed: 2018-1-2.

[23] Miller, M. Robust composition: Towards a unified approach to access control and concurrency control 2006. *Johns Hopkins: Baltimore, MD*, page 302.

[24] Miller, M. S., Samuel, M., Laurie, B., Awad, I., and Stay, M. (2008). Safe active content in sanitized javascript. *Google, Inc., Tech. Rep*.

[25] Miller, M. S., Yee, K.-P., Shapiro, J., et al. (2003). Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals.

[26] Moore, S., Dimoulas, C., Findler, R. B., Flatt, M., and Chong, S. (2016). Extensible access control with authorization contracts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 214–233. ACM.

[27] Moore, S., Dimoulas, C., King, D., and Chong, S. (2014). Shill: A secure shell scripting language. In *OSDI*, pages 183–199.

[28] Moore, S. D. (2016). *Software Contracts for Security*. PhD thesis.

[29] Parser Tools: lex and yacc-style Parsing. http://docs.racket-lang.org/parser-tools/. Accessed: 2018-1-2.

[30] Profile: Statistical Profiler. https://docs.racket-lang.org/profile/index.html. Accessed: 2018-3-23.

[31] Redell, D. D. (1974). Naming and protection in extendible operating systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC.

[32] Refinement Types in Typed Racket. http://blog.racket-lang.org/2017/11/adding-refinement-types.html. Accessed: 2017-12-24.

[33] Saltzer, J. H. (1974). Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402.

[34] Schoepe, D., Hedin, D., and Sabelfeld, A. (2014). Selinq: tracking information across application-database boundaries. In *ACM SIGPLAN Notices*, volume 49, pages 25–38. ACM.

[35] SQL As Understood By SQLite: CREATE TRIGGER. https://sqlite.org/lang_createtrigger.html. Accessed: 2018-1-2.

[36] SQLite. https://www.sqlite.org/. Accessed: 2017-12-31.

[37] Stiegler, M. (2007). Emily: A high performance language for enabling secure cooperation. In *Creating, Connecting and Collaborating through Computing, 2007. C5'07. The Fifth International Conference on*, pages 163–169. IEEE.

[38] Stiegler, M. and Miller, M. (2002). A capability based client: The darpabrowser. *Technical Report Focused Research Topic*.

[39] Strickland, T. S., Tobin-Hochstadt, S., Findler, R. B., and Flatt, M. (2012). Chaperones and impersonators: run-time support for reasonable interposition. In *ACM SIGPLAN Notices*, volume 47, pages 943–962. ACM.

[40] Tobin-Hochstadt, S. and Felleisen, M. (2008). The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406.

[41] Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., and Felleisen, M. (2011). Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM.

[42] Updateable and Insertable Views. https://dev.mysql.com/doc/refman/5.7/en/view-updatability.html. Accessed: 2018-1-2.

[43] Yang, J., Hance, T., Austin, T. H., Solar-Lezama, A., Flanagan, C., and Chong, S. (2016). Precise, dynamic information flow for database-backed applications. In *ACM SIGPLAN Notices*, volume 51, pages 631–647. ACM.