Fine-Grained, Dynamic Access Control for Database-Backed Applications

Ezra Zigmond
Harvard University
Cambridge, MA, USA
ezigmond@acm.org

ABSTRACT

Flaws in access control checks in database-backed applications frequently lead to security vulnerabilities. I present a new language, ShillDB, for writing secure, database-backed applications. ShillDB supports writing declarative database security policies as part of program interfaces, and the language runtime enforces these security policies.

CCS CONCEPTS

• Information systems → Relational database query languages; **• Security and privacy → Database and storage security**; **Software security engineering**; **• Software and its engineering → Interface definition languages**;

KEYWORDS

contracts, capabilities, language-based security

ACM Reference Format:
Ezra Zigmond. 2018. Fine-Grained, Dynamic Access Control for Database-Backed Applications. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion)*. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3191697.3213791

1 INTRODUCTION

Database-backed applications require fine-grained, dynamic restrictions on data access. For example, the information that a multi-user web application can display depends on the currently logged-in user. These restrictions are typically enforced by security code which sits between the database management system (DBMS) and the rest of the application code. However, hand-writing security checks seems to be difficult and error-prone, as broken access control is a common bug in web applications [11].

One can also enforce security policies at the DBMS level using security tools provided by the database, but this makes it cumbersome to give different privilege levels to different application components (which may be desirable if, for example, some application components came from untrusted third parties).

To address these limitations of current approaches to database access control, 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.[1]

2 CONTRACTS & CAPABILITIES FOR SECURITY

ShillDB builds on recent work in language-based security which has demonstrated that contracts can be used together with capabilities to enforce security policies at program interfaces [9, 10].

Contracts [4, 7] are a language feature that allow programmers to write specifications on values, such as functions and objects. These specifications are checked during program execution. Since contracts do not need to be statically checkable, they can express more complex specifications than those permitted by typical static type systems.

A *capability* is an unforgeable token that identifies a resource (such as a file or a database table) and conveys the authority to perform some action(s) on that resource. *Capability-safe languages* limit the origins of capabilities, providing a basis for reasoning about the authority that programs have during execution [8].

3 DESIGNING APPLICATIONS IN SHILLDB

In ShillDB, access to database tables is only possible through *view capabilities* which represent a restricted window into a database. Programmers can derive new view capabilities from existing capabilities (e.g. using `where`) as well as fetch or manipulate a view capability's underlying data. Function interfaces can specify *contracts* on view capabilities which allow for fine-grained restrictions on how capabilities can be used.

Following the design of Shill [9], a ShillDB program consists of an *ambient program* and a *capability-safe program*. Ambient programs can create new view capabilities using *ambient authority* (the privilege of the user invoking the program). Capability-safe programs do not have ambient authority and can only derive new capabilities from capabilities they initially receive. It is thus possible to reason about the database privileges a capability-safe ShillDB program has just by looking at what capabilities it is passed and what the program's contract is.

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04...$15.00
https://doi.org/10.1145/3191697.3213791

[1]The implementation of ShillDB is available at https://github.com/ezig/shilldb

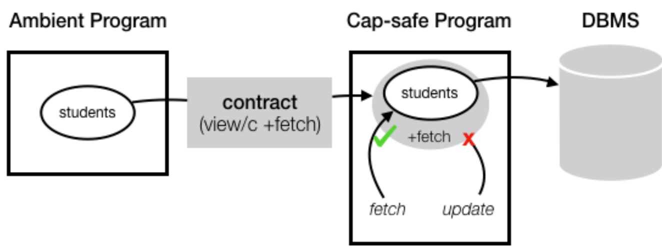

Figure 1: An overview of contracts and capabilities in ShillDB. The ambient program creates a capability for a `students` table, which is wrapped in a contract and passed to a capability-safe program. In this case, the contract only allows reading the table, so any operations invoked by the capability-safe program other than `fetch` will be rejected.

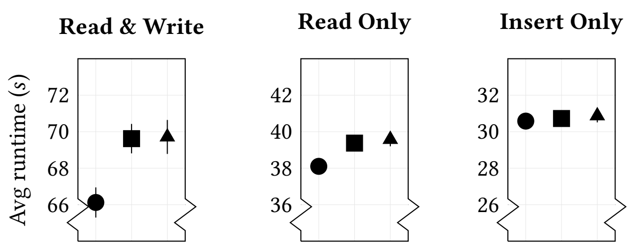

Figure 2: Average time required to run library reservation system workloads for the baseline (○), capability-based interface (□), and ShillDB implementation (△). 95% confidence intervals are indicated by vertical bars (but may be obscured by plotting symbols). Note that the y-axes begin at different values but the scale is consistent between plots. The overhead of the contracts and capabilities compared to the baseline is modest for all workloads (less than 5.4%).

To run a ShillDB program, a user invokes the ambient program. Figure 1 illustrates how ShillDB uses contracts and capabilities to provide security guarantees. Ambient programs can create capabilities for database tables and pass these capabilities to capability-safe programs. The interface of a capability-safe program applies contracts to capabilities. Within a capability-safe function, contracts act as *proxy objects*. These proxies intercept operations invoked on capabilities and can choose whether to forward an operation to the underlying capability (which will perform an action on the database) or to reject the operation.

ShillDB contracts enable a wide range of security policies, such as requiring that an inner join be performed on two tables before the contents of either table can be fetched, or only allowing the average value of a column to be viewed.

4 EVALUATION

To evaluate the usability and performance of ShillDB, I have used it to implement a library reservation system. ShillDB's contract system made it possible to express many realistic security policies for the library application. For example, contracts prevent users from deleting another user's reservations and allow users to see the total number of reservations for a book but not which users have reserved it.

Figure 2 shows the run time (averaged over 50 trials) for three different sequential request workloads for the library server. I ran the workloads using three different server implementations: one using Racket's [5] standard database interface as a baseline, one using ShillDB's capability-based database interface, and one using both the capability-based interface and contracts. The overhead of using both contracts and capabilities was at most 5.4% compared to the baseline.

5 RELATED WORK

Contracts have been used to enforce a variety of access policies [3, 6, 9]. Other systems have approached language-level database security by using static type systems to enforce access control policies [1, 2] or dynamic checks to enforce information flow policies [12]. ShillDB is the first to use contracts and capabilities for database access control.

ACKNOWLEDGMENTS

I am grateful to Stephen Chong and Christos Dimoulas for their helpful comments and advice. I thank Scott Moore for his feedback on an early version of this work.

REFERENCES

[1] Luís Caires, Jorge A Pérez, João Costa Seco, Hugo Torres Vieira, and Lúcio Ferrão. 2011. Type-Based Access Control in Data-Centric Systems. In *ESOP*. Springer, 136–155.
[2] Adam Chlipala. 2010. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications.. In *OSDI*. 105–118.
[3] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative policies for capability control. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*. IEEE, 3–17.
[4] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 48–59.
[5] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.
[6] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. 2012. Access permission contracts for scripting languages. *ACM SIGPLAN Notices* 47, 1 (2012), 111–122.
[7] Bertrand Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (1992), 40–51.
[8] M Miller. 2006. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. *Johns Hopkins: Baltimore, MD* (2006), 302.
[9] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. SHILL: A Secure Shell Scripting Language. In *OSDI*. 183–199.
[10] Scott David Moore. 2016. *Software Contracts for Security*. Ph.D. Dissertation.
[11] OWASP. 2017. OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[12] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, dynamic information flow for database-backed applications. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 631–647.