SeSQLite : Security Enhanced SQLite

Mandatory Access Control for Android databases

Simone Mutti
simone.mutti@unibg.it

Enrico Bacis
enrico.bacis@unibg.it

Stefano Paraboschi
parabosc@unibg.it

Università degli Studi di Bergamo, Italy
Department of Management, Information and Production Engineering

ABSTRACT

SQLite is the most widely deployed in-process library that implements a SQL database engine. It offers high storage efficiency, fast query operation and small memory needs. Due to the fact that a complete SQLite database is stored in a single cross-platform disk file and SQLite does not support multiple users, anyone who has direct access to the file can read the whole database content. SELinux was originally developed as a Mandatory Access Control (MAC) mechanism for Linux to demonstrate how to overcome DAC limitations. However, SELinux provides per-file protection, thus the database file is treated as an atomic unit, impeding the definition of a fine-grained mandatory access control (MAC) policy for database objects.

We introduce SeSQLite, an SQLite extension that integrates SELinux access controls into SQLite with minimal performance and storage overhead. SeSQLite implements labeling and access control at both schema level (for tables and columns) and row level. This permits the management of a fine-grained access policy for database objects. A prototype has been implemented and it has been used to improve the security of Android Content Providers.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls

Keywords

Android, Database security, Mandatory Access Control

1. INTRODUCTION

One of the clearest trends of the past few years has been the adoption by users of mobile portable devices, replacing personal computers as the reference platform for carrying out their daily activity.

The wide deployment of mobile operating systems has introduced a number of challenging security requirements, making explicit the need for an improvement of security technology. In the *bring your own device* (BYOD) scenario, for example, the need to separate the user data from the organization data is the core issue.

To achieve this goal, it appears crucial to assign a role to *Database Management Systems* (DBMS). Beside access and processing functionalities for defining, maintaining, and accessing the data stored in a *database*, each DBMS provides different features to ensure *data security*. Whenever a subject tries to access a data object, the access control mechanism checks the rights of the user against a set of authorizations, stated usually by the security administrator.

Unfortunately, current information systems often make a limited use of database access control facilities and embed access control directly in the application program used to access the database. This choice derives from the perceived difficulty in keeping the database users aligned with the user population in the application and from the flexibility obtained in the construction of the application thanks to the absence of access control restrictions to the database. (Our expectation is that this is going to change, but it will take a long time.) The management of access privileges outside of the database is instead the only available option in Android. In fact, access to the data managed by the SQLite database is controlled only at a level of service invocation by applications, relying on the *Android Permission Framework* in conjunction with *Content Providers*. Although widely used, this approach has several disadvantages, e.g., all security policies have to be implemented into each of the applications built on top of the data.

To overcome this limitation modern DBMSs provide their own authorization mechanisms in SQL which permit access control at the level of tables, columns or views. Only few DBMSs such as Oracle [1] and PostgreSQL [2] provide access control at the level of single tuples.

However, they use a permission model that is similar to, but separate from, the underlying operating system permissions. The database administrator creates the users and grants access permission to various database capabilities, with the option to pass some of them to other users.

Mandatory access control (MAC) policies regulate accesses to data on the basis of predefined classification of subjects and objects in the system. A subject is granted access to a given object if and only if some order relationship, depending on the access mode, is satisfied by the access classes of the object and the subject. The use of MAC models provides a characteristic feature called system-wide consistency, because all the access control decisions are guaranteed to be compliant with the MAC policy.

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2818000.2818041

The paper presents SeSQLite: a *Security-Enhanced SQLite* database. SeSQLite uses the features provided by MAC models to provide fine-grained access control at both schema level (for tables and columns) and row level. This permits the definition of fine-grained access policy for database objects.

Outline The paper is organized as follows. Section 2 provides an overview of functionalities provided by *SQLite*, and introduces *SELinux* describing the role of a MAC model. Section 3 describes the challenges to integrate SELinux in SQLite. Section 4 presents SeSQLite both at schema level and at tuple level. In Section 5 we discuss SeSQLite implementation, the performance results and all the optimizations introduced in order to reduce the overhead to a negligible level. Section 7 illustrates how the use of SeSQLite can improve the security provided by Android Content Providers. Section 8 provides a comparison with previous work in the area. Finally, Section 9 draws a few concluding remarks.

2. BACKGROUND

SQLite is an in-process library that implements an SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. It offers high storage efficiency, fast query operation, ACID transactions, small memory needs and no setup or administration. A complete SQLite database is stored in a single cross-platform disk file.

The core of the SQLite infrastructure contains the *user interface*, the *SQL command processor*, and the *abstract machine* [3]. The user interface consists of a library of C functions and structures to handle operations such as initializing databases, executing queries, and looking at results. The command processor acts exactly like a compiler: it contains a tokenizer, a parser, and a code generator. Essentially, the command processor outputs a *program* in an intermediate language. The *program* is generated for an abstract machine implemented in the SQLite library.

2.1 SELinux

SELinux is one of the most widely adopted implementations of Mandatory Access Control. It is also integrated in Android since version 4.2.

SELinux policies are expressed at the level of *security context* (also known as *security label* or just *label*). SELinux requires a security context to be associated with every process (or subject) and resource (or object), which is used to decide whether access is allowed or not as defined by the policy. Every request that a process generates to access a resource will be accepted only if it is authorized by both the classical DAC access control service and by the SELinux policy. The advantages of SELinux compared to the DAC model are its flexibility (the design of Linux assumes a *root* user that has full access to DAC-protected resources) and the fact that process and resource labels can be assigned and updated in a way that is specified at system level by the SELinux policy (in the DAC model, owners are able to fully control the resources). SELinux uses a closed world assumption, so the policy has to explicitly define rules to allow a *source* (the process) to perform a set of *actions* on a *target* (the resource). The rule also specifies the class of target on which the rule has to be applied (e.g., file, directory). An SELinux rule has the following syntax:

```
allow source_t target_t:class {actions};
```

3. CHALLENGES

Prior to our work, the challenges for integrating SELinux and SQLite were manifold. To the best of our knowledge, SeSQLite is the first work that shows an integration of a Mandatory Access Control in SQLite.

In this Section, we describe the differences between SQLite and modern RDBMSs, focusing on why the approaches used in modern RDBMSs are not compatible with the SQLite design and how Android tries to mitigate these limitations by the introduction of the *Content Provider*.

3.1 SQLite vs. Modern RDBMSs

Most large-scale database systems, like *PostgreSQL* and *Oracle*, have a large server package that provides the full database engine. A database instance consists of a large number of files organized into one or more directories on the server filesystem. In order to access the database, all of the files must be present and correct. All of these components require resources and support from the host such as dedicated service-user accounts, startup scripts, and dedicated storage.

In contrast, SQLite has no separate server. The entire database engine library is integrated into the application that needs to access a database. The only shared resource among applications is the single database file. By eliminating the server, a significant amount of complexity is removed. This simplifies the software components and nearly eliminates the need for advanced operating system support. Unlike a traditional RDBMS server that requires advanced multitasking and high-performance inter-process communication, SQLite requires little more than the ability to read and write to the file system.

Beside access and processing functionalities, currently, each RDBMS provides different features to ensure *data security*. Modern database systems use a permissions model that is similar to, but separate from, the underlying operating system permissions. The database administrator grants access permissions to various database capabilities. For example, *PostgreSQL* provides a complete set of authorization and authentication mechanisms.

SQLite, instead, is not a multi-user database, which means that anyone who has direct access to the file can read the database content. SQLite does not provide any kind of access control mechanism, it only provides a few proprietary extensions (e.g., database encryption).

3.2 Android and SQLite

In the Android platform, SQLite is used to manage system and user databases storing several types of information, like contacts, SMS messages, and web browser bookmarks. The access to SQLite databases is mediated by *Content Providers*. The Content providers are daemons that provide an interface to the SQLite library, for sharing information with other applications.

Android provides distinct security mechanisms at different layers to protect the data inside a SQLite database. At the application layer, the *Android Permission Framework* in conjunction with *Content Providers* provides access control to the data managed by the database but only at the level of service invocation by applications (see Figure 1). The access control model assumes that apps specify in their manifest the set of privileges that will be required for their execution.

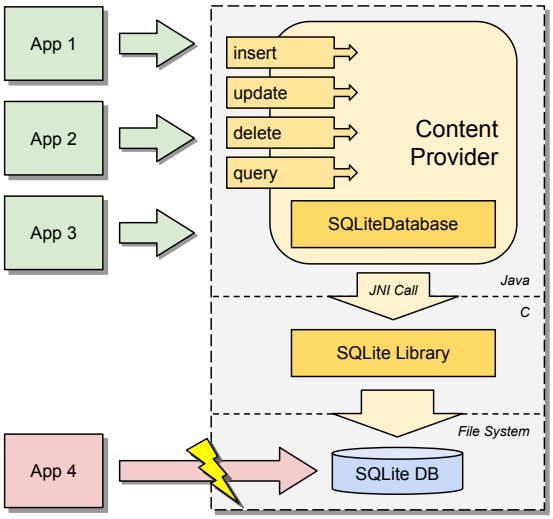

Figure 1: Content provider and SQLite interoperability.

Content Providers can also enforce permissions programmatically: the Content Provider code that handles a query can explicitly call the system's permission validation mechanism to require certain permissions.

EXAMPLE 1. *An app must hold the READ_CONTACTS permission in order to execute* READ *(i.e., SELECT) queries on the* Contacts *Content Provider.*

At the Linux kernel level, Android provides both *DAC* and *MAC* access control [4]. The former enforces security by means of user identifiers (*uid*) and group identifiers (*gid*); only the owner of the data (i.e., the Content Provider) holds the *r/w* permissions on the file. The latter provides support for the use of SELinux [5] into the Android operating system. Both DAC and MAC are designed to provide protection against an attempt to directly access the database by a process that it is not the Content Provider.

Due to the fact that SQLite is not a multi-user database (i.e., anyone who has direct access to the file can read the database content), the use of Discretionary Access Control (DAC) alone as security mechanism is not adequate, because significant weaknesses remain. For example, the granularity of the DAC permissions is too coarse, and there is the inability to confine any system daemons or *setuid* programs that run with the *root* or *superuser* identity.

Due to these limitations, the file-level granularity provided by DAC and MAC is not enough if we want to provide a fine grained access control over SQLite databases. The introduction of SeSQLite improves the definition and enforcement of the security requirements associated with SQLite databases.

4. RATIONALE OF THE APPROACH

Our proposal is to extend SQLite and integrate it with SELinux in order to provide fine-grained mandatory access control. By placing access control at the database level, one can ensure that access control policies are consistently applied to every user and every application.

This section discusses how the challenges described in the previous section were overcome in SeSQLite in order to enable the use of SELinux in SQLite. Overcoming these challenges requires changes and new additions to the library, and the creation of a new policy configuration.

4.1 The SeSQLite extension

SeSQLite was implemented as a *SQLite extension* to satisfy the following requirements:

R.1 - Backward Compatibility SeSQLite is designed to maintain backward-compatibility with common SQLite databases (i.e., no modification to the SQL syntax);

R.2 - Flexibility SeSQLite is designed to provide everything needed to successfully implement a Mandatory Access Control module, while imposing the fewest possible changes to SQLite. Moreover, it is designed to be easily adapted to different implementation of MAC (e.g., *SELinux* or *SMACK*);

R.3 - Performance SeSQLite must keep negligible the overhead on computational time and database size.

4.2 Access Control Granularity

As described in Section 3, in a SQLite database there are two different types of SQL object, at different granularity: the *Schema Level* and the *Tuple Level* (or *Row Level*).

In the following, the SQL statement compilation/execution workflow will be used as guideline and the introduction of security extensions will be presented when needed.

4.2.1 Schema Level

All SQL statements must be compiled before their execution. The compilation process usually involves four phases: (i) *Syntax check*, (ii) *Semantic check*, (iii) *Expansion* and (iv) *Code generation*.

The Syntax and Semantic checks control if the query can be "interpreted" (e.g., all keywords are present, all table names are spelled correctly). After these preliminary checks, an expansion phase is usually needed. For example, the "*" symbol is replaced with all the attributes' name of the table, the database that contains the table is selected.

The Syntax, Semantic checks and Expansion phase belong to the *parse* phase. At the end of this phase all the tables and attributes that have to be accessed are known and thus we can verify whether a user can execute or not the query.

EXAMPLE 2. *Consider a user who wants to perform the following query (see Figure 2):*

```
SELECT Type, Country, City FROM
    Address WHERE Contact_ID=2;
```

The statement accesses three columns within Address *table. The* Type, Country *and* City *attributes appear in the target list directly as a part of the query. The* Contact_ID *is used in the WHERE clause.*

Using this information, SeSQLite introduces a *schema check* to control if the query is allowed (i.e., the user can access all the tables and columns specified in the query) according to the SELinux policy. An error is immediately raised if the user does not have all the privileges to perform the query. According to the SELECT statement in Example 2, the user needs to hold at least the *select* privilege on the *Address* table and on the *Type*, *Country*, *City* and *Contact_ID* attributes. The same approach can be applied to *INSERT*, *DELETE* and *UPDATE* queries.

SELinux requires a security context to be associated with every process (subject) and object in order to decide whether

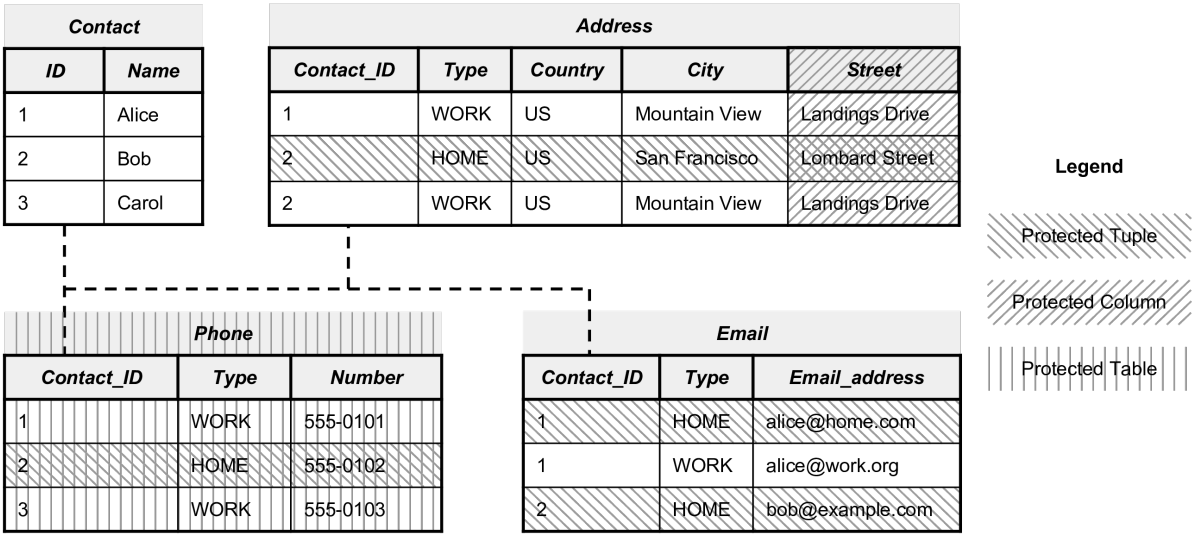

Figure 2: Example of a contacts database.

access is allowed or not as defined by the policy. To be compliant with this approach, SeSQLite introduces a new internal table, named *selinux_context*, used to store the security context associated with tables, views and attributes. For security reasons, the *selinux_context* table can not be directly modified by the user with SQL commands. Section 5 discusses how the schema level contexts can be modified.

If the schema check is successful, the code generator produces virtual machine code, aiming at the lowest cost execution plan, which will perform the work that the SQL statement requests (see Figure 3).

4.2.2 Tuple Level

The code generated by the previous phase is executed by an internal abstract machine, called *Virtual DataBase Engine* (VDBE). The abstract machine implements a computational engine specifically designed to manipulate databases.

The actions performed by the abstract machine can be summarized as (i) access to a database table, (ii) loop over each row, then (iii) clean up and exit. The loop is composed by the load of data from an attribute of the current row and their placement in a *ResultRow*, which represents the result set of the query.

At tuple level the query is always "satisfied". In fact, tuple level access control performs as a *filter* that automatically excludes any unaccessible tuple from the table scan. This allows SeSQLite to process only the tuples that can be accessed by the user, according to the action requested.

One way to enforce this approach is to use *authorization view* [1], but this could increase the cost and complexity of application development. An alternative approach is to allow queries to be written against database relations, but to modify the query by replacing the database relations with the view of the relation that is available to the user. For example, the *Virtual Private Database* (VPD) feature of Oracle's database server [6] implements fine-grained access control using query rewriting. Essentially this can be compared to automatically append conditions to a SQL query's WHERE clause as it executes, and dynamically changing the result returned by the query.

EXAMPLE 3. *Consider a user that wants to perform the following query:*

```
SELECT Type, Email_address FROM Email
    WHERE Contact_ID=1;
```

If we do not introduce tuple-level checks the result will be the following:

Type	Email_address
HOME	alice@example.com
WORK	alice@example.org

However, if we consider the same query and we want to enforce that a user can select only non protected tuples, with query rewriting the query becomes:

```
SELECT Type, Email_address FROM Email
    WHERE Contact_ID=1 AND check_tuple();
```

The check_tuple() function checks if the tuple is protected or not. The output of the query is the following:

Type	Email_address
WORK	alice@example.org

Query rewriting is the most common mechanism used to provide fine-grained access control at tuple level because the modifications are internal to the database and do not require any adaptation at application level.

SeSQLite uses a modified version of traditional query-rewriting. Due to the fact that SQLite does not provide a multi-user database, the decision to allow or not the access to a tuple should be taken using another piece of information, i.e., the security context. At schema level the problem of labeling is solved by adding an additional table used to store the label associated with a table or an attribute. At tuple level, the label is stored through the use of an extra attribute added to each table, named *security_context*. The *security_context* column gives users a method to access the security context of every tuple and allows the use of SELinux in the access control mechanism.

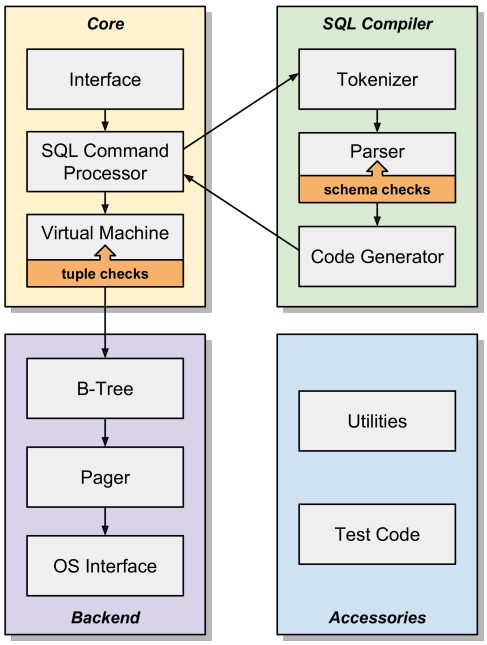

Figure 3: SQLite Architecture with SELinux checks.

This design gives us a characteristic feature called system-wide consistency in access control, because all the access control decisions are made by the SELinux security server based on a single declarative policy.

5. IMPLEMENTATION

In this section we discuss the key aspects involved in the implementation of SeSQLite. SeSQLite stems from SQLite version 3.8.6[1] and will be released under an open source license as described in Section 11. The modifications to the SQLite library meet the security requirements specified in Section 4. They can be structured into the following activities: (i) initialize the security context in a SQLite database, (ii) support for schema level check, (iii) support for row level check, (iv) optimizations.

5.1 Extension Hooks

As explained before, in order to comply with *R.2 - Flexibility*, we kept the changes to the SQLite library to the minimum, so we didn't integrate the SELinux checks directly in the SQLite code. As shown in Figure 4, we used some already available extension hook and implemented some others, so that our work can be used also by other extension to accomplish different tasks.

The following hooks were already available:

sqlite3_set_authorizer provides a skeleton to perform schema level checks as described in Section 5.3;

sqlite3_commit_hook (**sqlite3_rollback_hook**) registers a C function to be executed every time that a commit (rollback) is performed; This hooks are used to empty the security decision cache as discussed in Section 5.5.2

We also implemented the following new hooks:

sqlite3_create_pragma registers a new *PRAGMA* in the current SQLite session. The name of the created *PRAGMA* cannot conflict with the other ones already registered (both the system ones and the custom ones);

sqlite3_before_create_table_hook executes a C function *before* the creation of a table. This hook is used to prepare the table for schema level checks;

sqlite3_after_create_table_hook executes a C function *after* the creation of a table. This function is used to finalize the modification applied to a just created table;

sqlite3_query_insert_hook executes a C function during an INSERT operation;

sqlite3_query_rewrite_hook executes a C function during the compilation of a query appending a new "node" (see Section 5.5) in the WHERE clause;

sqlite3_schemachange_hook registers a callback function that is invoked every time a schema change occurs (e.g., ALTER and RENAME table);

sqlite3_set_xattr ()**sqlite3_set_xattr**) set (get) an extended attribute (e.g., the source security context).

Furthermore, we adapted the *initialization* process in order to assign a security context to every object in the database.

5.2 SeSQLite Security Contexts

SQLite follows the *lazy loading* pattern, which means that the initialization is deferred until the user performs a SQL primitive (e.g., create). Opening a database does not trigger the initialization process. The initialization process consists in the in-memory materialization of two tables; *sqlite_master* and *sqlite_temp_master*. The former defines the schema for the database (e.g., tables, views), the latter works just like *sqlite_master* except that it is meant for temporary tables, indices and triggers. Both tables are read-only. No one can change these tables using UPDATE, INSERT, or DELETE. The tables are automatically updated by CREATE and DROP commands.

As explained in Section 4, the row level security context is maintained using a persistent attribute in each table. To maintain a label mapping for the schema level that is compliant with *R1 - Backward Compatibility* and *R2 - Flexibility*, we must not change either the SQL syntax understood by SQLite [7], or the file format [8]. A new "internal" table named *selinux_context* has been introduced. This table works like the *sqlite_master*, maintaining the labels regarding all the SQL "schema" objects in a database. Table 1 shows an example of the *selinux_context* content.

The label mapping is initialized during the in-memory materialization process using *PRAGMA* statements. The PRAGMA statement is a SQL extension specific to SQLite and used to modify the behavior of the SQLite library or to query the SQLite library for internal (non-table) data. The PRAGMA statement is issued using the same interface as other SQLite commands (e.g., SELECT, INSERT). Specific PRAGMA statements were implemented in SQLite on an as-needed basis, through the use of the *sqlite3_create_pragma* hook. Due to space constraints we do not describe the PRAGMA statements added by the SeSQLite extension.

5.2.1 SQL objects labeling

A configuration file, named *sesqlite_contexts*, was introduced, to assign an initial contexts to SQL objects.

The items in the *sesqlite_contexts* file declare the security contexts applied to databases, tables, views, columns and tuples when the database is initialized. The file uses regular

[1]This version is the one used in Android 5.1.0

security_context	database_name	table_name	column_name
u:r:db_data_table_t:s0	main	sqlite_master	-
u:r:db_data_column_t:s0	main	sqlite_master	type
u:r:db_data_column_t:s0	main	sqlite_master	name
u:r:db_data_column_t:s0	main	sqlite_master	tbl_name
u:r:db_data_column_t:s0	main	sqlite_master	rootpage
u:r:db_data_column_t:s0	main	sqlite_master	sql

Table 1: Table used to store selinux contexts for schema objects.

Class	regex	security_context
db_database	*	u:r:sqlite_db_t:s0
db_table	*.Email	u:r:db_data_table_t:s0
db_view	*.*	u:r:db_data_view_t:s0
db_column	*.Email.*	u:r:db_data_column_t:s0
db_tuple	*.*	u:r:db_data_tuple_t:s0

Table 2: Snippet of sesqlite_contexts file.

	Size	Overhead
SQLite	5 123 KiB	-
SeSQLite (schema level)	5 127 KiB	+0.08%
SeSQLite (row level)	5 254 KiB	+2.56%
SeSQLite (full)	5 256 KiB	+2.6%

Table 3: Size comparison between SQLite and SeSQLite based on 50000 INSERT.

expression (regexp) to match for example a database, a table or a set of tables with a specific SELinux label (Table 2 shows an example). The rule used to match an element is the *"most specific takes precedence"*.

SeSQLite has a namespace hierarchy where a database is the top level object, followed by tables and then columns. This hierarchy is supported as follows (Table 2 shows an example af an *sesqlite_contexts* file):

db_database is used to specify the security context for database objects. Due to the fact that databases are the top level object, the pattern to match is composed by one element, the database name.

db_table is used to specify the security context for table objects. The pattern to match is composed by two elements. The former is used to identify the database name and the latter is used to identify the name of the table. In Table 2 the meaning of line 2 is the following: "the security context associated with the *Email* table contained in any database is *u:r:db_data_table_t:s0*". The same approach is used to assign the security context to view objects;

db_column is used to specify the security context for column objects. Here, the pattern to match is defined by three elements. First the database name, second the table name and third the column name. In Table 2 row 4 means: "in any database, all the columns of the *Email* table are labeled as *u:r:db_data_column_t:s0*";

db_tuple is used to specify the security context for tuple objects. The syntax is similar to the one used by *db_table*. The pattern to match is composed by two elements. The database name and the table name. This means that all the tuples inserted in a table will be assigned the specified default *security_context*.

5.3 Schema level

Every SQL statement must be compiled into a VDBE program before its execution using either *sqlite3_prepare* or *sqlite3_prepare_v2*.[2]

[2]In the remaining of the document we will focus our analysis only on *sqlite3_prepare_v2*, however all the considerations done for this function can be extended to the others.

To implement the approach showed in Section 4 an *authorizer* callback was implemented via *sqlite3_set_authorizer*. At various points during the compilation process the authorizer callback is invoked to see if those actions are allowed. The authorizer callback should return SQLITE_OK to allow the action, SQLITE_IGNORE to ignore the specific action but allow the SQL statement to continue to be compiled, or SQLITE_DENY to cause the entire SQL statement to be rejected with an error. The workflow followed by the authorizer is the following: (i) initialize the additional in-memory structure (ii) retrieve the security context associated with the SQL object, (iii) decide if the user has the privilege to access the SQL object and (iv) clean the in-memory structure and exit.

The first step is used to initialize all the additional structures that will be used during the evaluation of the query. For example, due to performance reasons we implemented a user-space access vector, see Section 5.5.2.

The second step permits to retrieve the security context associated with the tables or attributes used in the query. As explained in Section 4, this information is stored in an "internal" table. More specifically, we implemented the *security_context* table as a *Virtual Table*.

From the perspective of an SQL statement, the virtual table object looks like any other table or view. But, queries and updates to a virtual table invoke callback methods on the virtual table object, instead of reading and writing to the database file. The virtual table mechanism allows an application to publish interfaces that are accessible from SQL statements as if they were tables. SQL statements can in general do anything to a virtual table that they can do to a real table, with few exceptions (e.g., it is not possible to create a trigger on a virtual table).

In this way we can provide both low latency access to the security contexts and persistence. The former is achieved using an in-memory data structure, i.e., a hash table, filled during the initialization process. To comply with requirement *R.1 - Backward Compatibility*, we used a real table to store the mapping between tables/attributes and security contexts. Table 3 shows the memory overhead, in terms of size, introduced by the *security_context* table.

In the third step the information retrieved by the previous steps is sent to the SELinux security server to be checked.

When *sqlite3_prepare_v2* is used to prepare a statement, this might be re-prepared during *sqlite3_step* due to a schema change. *sqlite3_step* function is called multiple times at execution time. SeSQLite ensures that the correct authorizer callback remains in place during all these steps.

5.4 Tuple level

The approach used to implement tuple-level checks differs from the one used at schema-level for obvious reasons. It is unfeasible to create an additional table to maintain the mapping between each tuple and its own security context. Hence, we decided to add a column to each table in order to directly store the security context within the tuple. The hooks *sqlite3_before_create_table_hook* and *sqlite3_after_create_table_hook* have been created to implement this mechanism.

As explained in Section 4, query rewriting is a powerful mechanism to enforce tuple-level access control, but not all the SQL actions can be controlled using this strategy. For example, we cannot rewrite an INSERT query for two reasons: (i) to be compliant with requirement *R.1 - Backward Compatibility* (i.e., do not modify the INSERT statement with the addition of new "constructs") and (ii) the INSERT syntax does not allow the use of a WHERE clause, thus we cannot append a constraint that enforces the access control.

The *sqlite3_insert_hook* has been implemented to overcome this limitation. It registers a callback function that is invoked prior to each INSERT operation. This way we can perform access control and add the right security context.

For the others actions, such as UPDATE, DELETE and SELECT we used a query rewriting strategy. Due to performance overhead two different query rewrite approaches was proposed and evaluated (see Section 6.1).

***selinux_check_access* SQL function** The first approach used to implement the query rewrite is based on a custom SQL function, named *selinux_check_access*. It represents a wrapper of the namesake function exposed by the SELinux library. This function is appended to the *where clause*, checking for each row if the process that is executing the query has the requested access on each tuple (based on its security context);

SQL IN operator The second approach leverages the use of the SQL "IN" operator. The idea is that given a specific action (e.g., select, update) SeSQLite computes the set of security context used in the database for which the process has the requested access. The allowed security contexts are enclosed in an "IN" operator which is appended to the *where clause*. Under specific circumstances this approach is faster than the previous one (see Section 6.1 for more details).

5.5 Optimizations

In order to comply with the minimal overhead requirement expressed in *R.3 - Performance* we introduced some optimizations in the code. Here we discuss the most interesting ones.

5.5.1 Context Translator

The performance overhead imposed by our checks can be further lowered using integers to identify security contexts instead of strings. The benefit of this change is twofold:

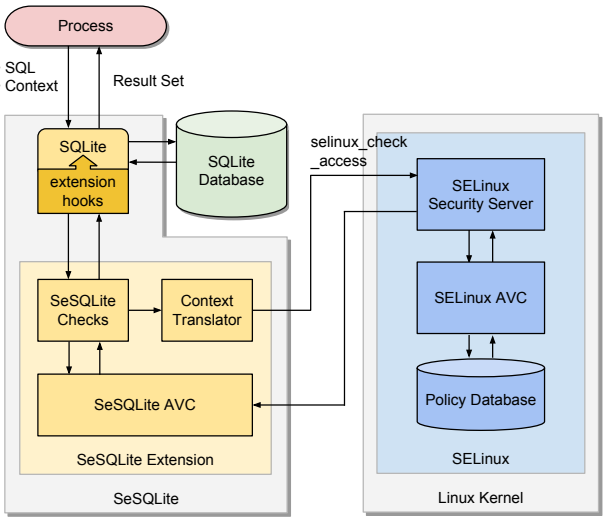

Figure 4: SESQLite architecture overview.

- Reduces the SeSQLite memory footprint which, as shown in Table 3, is mostly attributable to the addition of the extra *security_context* column to every table used to perform row level access control;

- Minimizes the performance overhead reducing the key size in the SeSQLite AVC that will be discussed in Section 5.5.2.

The Context Translator stores the mapping between the integer id and the security contexts that it represents. All the internal operations are performed on the ids, but when SeSQLite needs to talk to the outside world (e.g., initialization of the security contexts from the *sesqlite_context* file), the context translator provides and stores the correct translation.

When a translation is not available, the context translator is responsible to assign a new id to the security context and make the mapping persistent. In order to comply with *R.1 - Backward Compatibility*, the table *selinux_id* keeps the mappings in the database file. Any modification to this table is denied by the authorizer except the modifications coming directly from the translator itself.

5.5.2 SeSQLite Access Vector Cache

The SELinux security server can be queried using the system call *selinux_check_access*, which checks whether the *source* context has the *access permission* for the specified *class* on the *target* context. The SELinux architecture embeds an *access vector cache* (AVC) component that caches the access decisions already computed based on the policy database. The AVC minimizes the performance overhead of SELinux access control. [9]

Nevertheless, when SeSQLite invokes the in-kernel SELinux security server, it needs to perform a context switch, which is a heavy operation. So, it is crucial to minimize the number of system call invocations in order to reduce the performance overhead given by the additional security checks. This is especially true in row level access control. A single query can fetch a substantial number of tuples so, without additional measures, we may invoke a system call for each row.

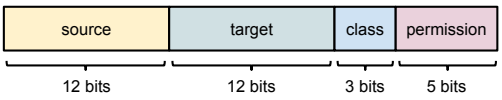

Figure 5: 32 bit integer used as dictionary key.

In a common SELinux policy, a sizable number of objects tend to share a small number of security contexts. When the combination of security contexts and action is the same, the equal result shall be returned, so we can cache the decisions in a user-space SeSQLite AVC, which, using the security ids in place of the security contexts, will also reduce the performance overhead. When the result is not already cached by the SeSQLite AVC, SeSQLite uses the *Context Translator* to map the security id to their counterparts before invoking the system call *selinux_check_access*. The experimental result of introducing the SeSQLite user-space AVC is described in the following table.

	kernel space (µs)	**user space** (µs)
cache miss	48.25 ($\sigma = 2.76$)	48.28 ($\sigma = 2.42$)
cache hit	28.96 ($\sigma = 1.87$)	0.24 ($\sigma = 0.18$)

In order to further compress the key used in the hash table, we decided to store the quadruplet composed by *source*, *target*, *class* and *permission* in a single 32-bit integer composed as shown in Figure 5.

Using this compression it is still possible to use 4096 different sources and targets, 8 classes and 32 permissions. At the time of writing the SELinux reference policy [10] defines 6 classes and 21 permissions (of which only 4 classes and 10 permission and relevant for the case of a simple database such as SQLite).

The SeSQLite AVC is cleared after each SQL primitive execution. In this way we can guarantee the correctness of transactions. In fact, without an internal caching mechanism the access level of the process might be changed in the policy while a query is being executed, producing a result set with an inconsistent access level.

6. EXPERIMENTAL RESULTS

One of the most appreciated features of SQLite is its performance. This is particularly crucial since SQLite plays a central role in some modern operating system (e.g., Android). So, a clear requisite of a SQLite extension is to have minimal overhead in terms of performance.

A performance test suite named *speedtest1* [11] is already included in the SQLite source tree to benchmark the library. The *speedtest1* utility is composed by some common and uncommon SQL operations. They span from simple a SELECT to a four-ways JOIN.

For the evaluation of the performance impact of the techniques presented in this paper, we executed a series of experiments based on this utility comparing the base SQLite library with both an optimized version of SeSQLite and a non-optimized one. The most important optimizations are described in Section 5.5. The experiments were run on a PC with Intel i7 3.4GHz/L3-4MB processor, 16GB RAM, 240GB SSD and Fedora 21 [3].

[3]See section 11 on how to obtain the performance evaluation on Android devices.

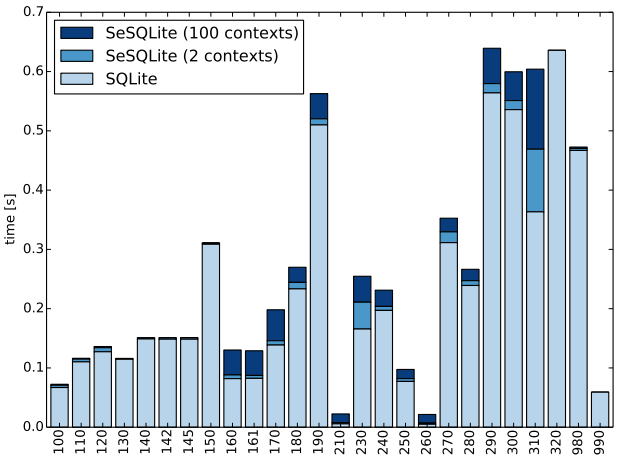

Figure 6: Comparison of the CPU time overhead between the base SQLite library and the *optimized* version of SeSQLite (avg. result of 100 *speedtest1* run).

	time	**overhead**
SQLite	5.846s	–
SeSQLite (2 contexts)	6.132s	+4.8%
SeSQLite (100 contexts)	6.741s	+15.3%
SeSQLite no-opt	13.052s	+123.23%

Table 4: Total CPU time and overhead for the test suite *speedtest1* executed using SQLite and SeSQLite.

Table 4 shows the comparison between SQLite and both non-optimized and optimized version of SeSQLite. The non-optimized version shows a significant overhead (+123.23%), while the optimized version shows a minimized overhead ranging from (+4.8%) using 2 security_contexts to (+15.3%) using 100 security_contexts.

The test presenting the biggest overhead is the 310, which is a four-ways join. We have to check the access privileges for four tables, but the overhead is still reasonable[4].

Figure 7 shows the overhead introduced by the basic SQL operations in SeSQLite. The overhead remains negligible even when the number of tuples grows.

6.1 Query rewriting comparison

As discussed in Section 5.4, we implemented two different types of query rewriting to deal with tuple level access control: (i) the use of *selinux_check_access*, (ii) the use of *SQL IN* operator. We evaluated the overhead imposed by the different approaches when used with *speedtest1* and the findings are shown in Figure 8. On one hand, it is clear that the overhead imposed by the *selinux_check_access* approach remains fixed even in the presence of a bigger number of different security contexts loaded in the database. The *selinux_check_access* function is computed for every tuple at query run time, so the overhead depends on the number of tuples but not on the number of security contexts.

On the other hand, the *SQL IN* operator has to creates a list of accessible security contexts and check if the one assigned to each tuple is contained in it. While this approach only has to perform SELinux checks when building the list

[4]The pairing between the test ids can be found in the *speedtest1* utility [11].

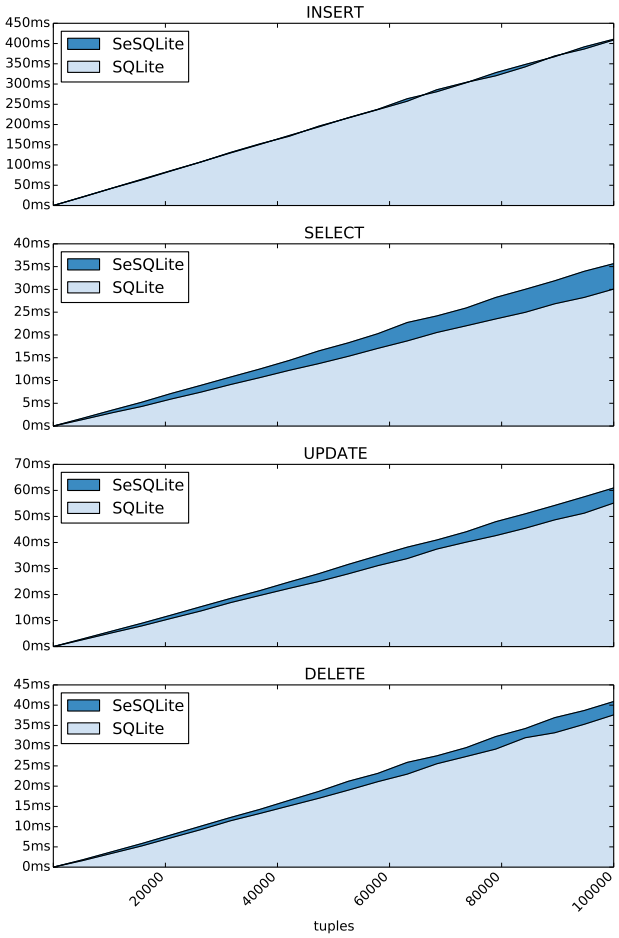

Figure 7: Comparison of the CPU time overhead between the base SQLite library and the *optimized* version of SeSQLite during INSERT, SELECT, UPDATE and DELETE operations.

at compile time, the overhead depends on the number of security contexts in the "IN" operator (the bigger the list, the more time spent on checking the tuple's security context inclusion). A significant overhead gap emerges between 7 and 8 security contexts (due to the fact that the data structure that contains the list needs to be resized).

In order to minimize the global SeSQLite overhead, at compile time we build the list for the *SQL IN* approach, but when the list grows over the 7th element, we switch to the *selinux_check_access* approach. This allows SeSQLite to always benefit from the best approach, keeping the overhead to a mere $\sim 5\%$ when the number of security contexts is low (the usual case) and to limit it at $\sim 13\%$ when the number of security contexts grows.

7. ANDROID INTEGRATION

We now provide a description of the challenges to enable the concrete use of SeSQLite in Android. The system has been implemented extending version 3.8.6 of SQLite and version 5.1.0 of AOSP.

The current SQLite implementation for Android spans different levels of the Android stack. At the *Application Framework* level, the *SQLiteDatabase* class provides access to the

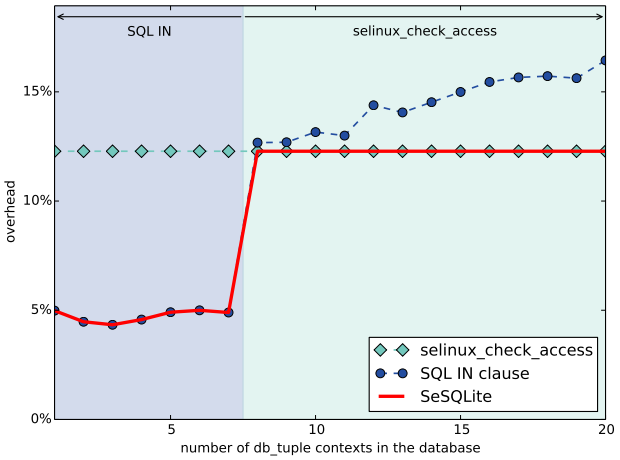

Figure 8: Comparison of the two query rewriting approaches presented in Section 5.4. When the number of SELinux contexts in the database is small, the SQL IN operator is used, otherwise the *selinux_check_access* function is used.

centralized Java Native Interface (JNI) bindings for SQLite interaction. The *android_database_SQLiteConnection.cpp* file is the JNI bridge. At the *Libraries* level, SQLite consists of the *libsesqlite* library, described in Section 5.

7.1 SQLiteDatabase

SQLiteDatabase is the base class for working with a SQLite database in Android and provides methods to *open*, *query*, *update* and *close* the database. If an app or a Content Provider wants to access a SQLite database it has to instantiate the SQLiteDatabase class and to use it to manage the database. Although in the current scenario this is enough to use SQLite, for the use of SeSQLite the *security context* of the caller is also needed.

This information cannot be retrieved by the library itself because, the library has the same context of the process that loaded it. This means that if we use this approach in the Content Provider scenario we will retrieve the security context associated to the Content Provider itself instead of the one associated to the app that issued the query. For this reason, a modification of the *SQLiteDatabase* class was needed. We introduce the *SESQLiteDatabase* class which allows to retrieve the security context of the caller process, both if the class is instantiated by a Content Provider or directly by a an app.

The *SESQLiteDatabase* class retrieves the security context of the caller by using (i) the *Binder.getCallingPid()* if the binder IPC is direct, or passed by other intermediate components in order to get the PID of the caller and then (ii) using the *SELinux.getPidContext()* to retrieve the security_context assigned to the process with that PID.

7.2 SeSQLite and Policy modularity

According to the approach used to assign security contexts to third party apps, SeSQLite assigns, during creation, to each tuple a default security context based on the information contained in the *sesqlite_contexts* file.

Although this approach brings several advantages, in emerging scenarios it suffers of several limitations. For example, we cannot distinguish tuples belonging to different

users and/or environment if we enable *Android for Work*. In fact, in the current implementation of Android (5.1.1), the management of different users and environment is done at application level. Android maintains a copy of each database (e.g., contacts list, download, bookmark) for each user/environment and the security mechanisms are provided by the *Content Providers* in conjunction with the DAC model, due to the fact that at SELinux level the different databases are assigned to the default security context. This is a significant limitation, since apps can get a concrete benefit from the specification of their own policy [12, 13, 14].

A step ahead in the direction of policy customization is represented by *Android M*. In the *M* release each user is assigned to a specific *category* building a *Multi Category Security* (MCS) model. MCS works like the DAC extended attributes. Users are assigned to categories and can apply these categories to their discretion to content that they own.

The introduction of categories brings several advantages in terms of flexibility. In Android M, through the categories, each user run apps with a distinctive security context, so SeSQLite labels the SQL objects differently per each user. With a single database file, through the use of SeSQLite the access can be customized per user in the policy. SeSQLite represents a step forward in this long-term vision.

8. RELATED WORK

SE-PostgreSQL [2] is a built-in enhancement of *PostgreSQL*, that provides additional access controls based on Security-Enhanced Linux (SELinux) security policy. This work inspired SeSQLite and guided its implementation. SE-PostgreSQL permits access control on both schema level and row level. Even if the objective is similar, the architecture is different. PostgreSQL is a real DBMS, with a dedicated process running in isolation, while SQLite is a server-less in-process library. For this reason the security constraints are different and this has a big impact on the implementation of the solution. For example, in PostgreSQL every element (e.g., tables, columns, tuples) is stored in the catalog jointly with its characteristics and a unique object-id. In order to label each object with its security context, a single additional column to the catalog is needed. Another architectural difference is that SeSQLite has been implemented as an extension that can be plugged into SQLite when requested, with limited changes on the base source code.

9. CONCLUSIONS

Security is correctly perceived, both by technical experts and customers, as a crucial property both of desktop and of mobile operating systems. The integration of SELinux into SQLite is a significant step toward the realization of more robust and more flexible security services.

The attention that has been dedicated in the SELinux initiative toward the protection of system components is understandable and consistent with the high priority associated with the protection of core privileged resources. Our approach is the natural extension of that work. It is to note that this design does not require to adapt applications that access the database. The approach can be considered an application of the "defense in depth" security principle, with a reduced criticality of the Content Provider, because the database itself is able to enforce security mechanisms.

The extensive level of reuse of SELinux constructs that characterizes the language demonstrates the flexibility of SELinux and facilitates the deployment of the proposed solution. The paper shows that the potential for the application of SELinux associated with SQL objects is quite extensive.

10. ACKNOWLEDGMENTS

This work was partially supported by a Google Research Award (winter 2014), by the Italian Ministry of Research within the PRIN project "GenData 2020" and by the EC within H2020 under grant agreement 644579.

11. AVAILABILITY

In the spirit of open science and open source, additional documentation and the *SeSQLite* source code are available at *http://unibg-seclab.github.io*.

12. REFERENCES

[1] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD*, SIGMOD '04. ACM, 2004.

[2] K Kohei. Security Enhanced PostgreSQL, 2013.

[3] SQLite - The Architecture of SQLite. http://sqlite.org/arch.html .

[4] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS 13)*, 2013.

[5] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, NJ, USA, 2006.

[6] Kristy Browder and Mary Ann Davidson. The virtual private database in Oracle9iR2. *Oracle Technical White Paper, Oracle Corporation*, 500, 2002.

[7] SQLite - SQL As Understood By SQLite. https://sqlite.org/lang.html .

[8] SQLite - The SQLite Database File Format. https://sqlite.org/fileformat.html .

[9] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux Security Module. *NAI Labs Report*, 1(43):139, 2001.

[10] SELinux Object Classes and Permissions Reference. http://selinuxproject.org/page/ObjectClassesPerms#Database_Object_Classes .

[11] SQLite - speedtest1. http://www.sqlite.org/src/finfo?name=test/speedtest1.c .

[12] Enrico Bacis, Simone Mutti, and Stefano Paraboschi. AppPolicyModules: Mandatory Access Control for Third-Party Apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 309–320. ACM, 2015.

[13] Simone Mutti, Enrico Bacis, and Stefano Paraboschi. Policy Specialization to Support Domain Isolation. In *Automated Decision Making for Active Cyber Defense (SafeConfig15-ACD)*. ACM, 2015.

[14] Simone Mutti, Enrico Bacis, and Stefano Paraboschi. An SELinux-based Intent manager for Android. In *IEEE Conference On Communications and Network Security*, Florence, Italy, September 2015.