An Empirical Study of Local Database Usage in Android Applications

Yingjun Lyu*, Jiaping Gui*, Mian Wan, William G.J. Halfond
Department of Computer Science
University of Southern California
Email: {yingjunl, jgui, mianwan, halfond}@usc.edu

***Abstract*—Local databases have become an important component within mobile applications. Developers use local databases to provide mobile users with a responsive and secure service for data storage and access. However, using local databases comes with a cost. Studies have shown that they are one of the most energy consuming components on mobile devices and misuse of their APIs can lead to performance and security problems. In this paper, we report the results of a large scale empirical study on 1,000 top ranked apps from the Google Play app store. Our results present a detailed look into the practices, costs, and potential problems associated with local database usage in deployed apps. We distill our findings into actionable guidance for developers and motivate future areas of research related to techniques to support mobile app developers.**

***Index Terms*—Mobile applications, database, energy, performance, security, empirical study.**

I. Introduction

The past decade has seen incredible growth in the amount of mobile apps available to end users. Developers compete for end users by designing innovative apps that can provide a rich user experience by combining data from web services, location services, and sensor data. In addition to innovating new services, developers also strive to create apps that are reliable, responsive, secure, and consume a minimal amount of limited resources, such as network data and energy. Balancing these concerns is important for developers as complaints about excessive resource usage or poor performance can lead to poor ratings for their apps [1], [2], [3].

Easy access to and management of data underpins important features in many mobile apps. To improve the responsiveness and reliability of their apps, developers frequently take advantage of local databases, such as SQLite, to store and manage their data. Local databases are stored directly on the mobile devices. They can be synchronized with remote databases and can help maintain reliable and responsive service even when the mobile device does not have a reliable connection. An application can also share its local data with other applications. These benefits have led to their widespread use and popularity; conservative estimates are that they are used in over 50% of all mobile apps.

Although local databases offer many benefits to developers, their use comes with potential problems. Some of these are similar to those that accompany remote database server usage. For example, both local and remote databases can be attacked using SQL injection (SQLI) attacks, if developers do not follow best practices for validating user input and interacting with the databases [4], [5], [6]. However, some of the problems are unique to the context in which mobile devices operate. Many database operations, such as transactions, require file locking and rollback capability. This means that the use of such operations is resource intensive, requiring CPU time, memory, and I/O access. Unsurprisingly, the use of these operations means that the use of local databases also comes with a high energy cost; recent studies have found that they are among the top two or three energy consumers of all services on mobile devices [7]. Lastly, despite the seemingly similar use of SQL, local databases often have different query syntaxes and semantics than traditional remote databases. This can mean that typical operations, such as batching multiple database queries, are handled differently on local and remote databases, potentially causing apps to behave incorrectly. Developers lack support for identifying when their code may suffer from these problems (e.g., SQLI and syntactical issues) and performance information (e.g., energy) that can help them design and implement more efficient apps.

Obtaining detailed, accurate, and complete information about local database usage poses many challenges. A naive approach, such as simply counting invocations to database related APIs in the source or Dalvik bytecode, is likely to only provide an incomplete picture. The reason for this is that many database APIs also allow developers to define queries as raw strings that contain the SQL queries to be executed by the database. These strings are dynamically constructed at runtime along the different paths of execution and may therefore define a range of different queries. These types of queries are widely used (see Section IV-C) and would require a path sensitive analysis to statically identify their potential runtime forms. Other kinds of useful information, such as energy and runtime, require sophisticated energy measurement platforms and instrumentation to isolate and measure their cost in an app. In related work [7], [8], only high-level (i.e., component level) energy information was provided for local database usage. Overcoming these technical challenges presents a significant barrier to obtaining accurate and complete information about the full range of local database usage in mobile apps.

In this paper, we present the results of an extensive study of database related behaviors in mobile apps. To overcome

* Yingjun Lyu and Jiaping Gui contributed equally to this work.

the significant challenges described above, we employed and adapted a range of dynamic and static program analysis techniques. These allowed us to more accurately identify queries, isolate runtime behaviors of specific APIs, and scale our study up to a large number of mobile apps. Our study addressed a range of questions about database usage, which included the types of queries used, performance costs, and the prevalence of certain problematic patterns of query construction. To the best of our knowledge, our study is the first to employ these forms of program analysis towards the goal of better understanding database usage in mobile apps, giving our study a unique perspective and range of results.

Our study reveals several interesting observations that provide concrete guidelines for app developers and motivate areas for future research work in the software engineering community. We found a large number of violations of many types of best practices regarding database usage in mobile applications. From a security perspective, we found that developers frequently use vulnerable APIs and patterns to execute and build SQL commands; we found that a large number of these instances were avoidable and could be refactored to prevent SQLI attacks by using secure APIs or parameterized queries. On top of this, we also found that many apps used untrusted inputs to build SQL commands, a practice that raises the risk of SQLI attacks. From the energy/performance perspective, we found that database initialization and write operations are the most expensive. We also found that, unfortunately, these operations appear frequently in loop structures; many of which are not properly batched in explicit transactions. This practice can lead to high resource consumption and cause significant inefficiencies. The results of our study provide developers with interesting insights into how databases are used in mobile apps and can help them to make better design and implementation decisions for their mobile apps.

II. Research Questions

Our investigation into database related usage in mobile apps is comprised of eight research questions (RQs). These broadly deal with three areas of interest for practitioners and researchers: usage patterns, security, and performance. We organize our presentation of the RQs in a top-down order by first addressing RQs at the application level then moving to lower levels, such as individual APIs and code constructs.

Our first research question is at the highest level and focuses on app-level database usage. It is **RQ 1: Which local databases are most widely used in mobile apps?**. In this RQ, we obtained a high-level picture of how frequently local databases are used in mobile apps and which types are the most widely used.

The next research question focuses on understanding the nature of the apps' interaction with their local databases. Namely, we are interested in information about the types of operations (e.g., transactions and queries) carried out by the apps' code. Formally, our RQ is **RQ 2: Which APIs are used to interact with the databases?**. In this RQ, we quantified the interactions of the apps with the local databases based on their invocations of database related APIs. The results of this RQ provide us with insights into the typical operations employed by app developers.

As we mentioned in the introduction, many database commands can be specified as raw strings that are passed as parameters to certain database APIs. These APIs are collectively referred to as Raw Query APIs (RQAs). An example of an RQA is `execSQL(java.lang.String)`. Since we found that these APIs are widely used, our next RQ focused on identifying the types of commands issued by these RQAs. More formally, our RQ is **RQ 3: What types of commands are issued as raw queries?**. For this RQ, we analyzed the RQA to identify the potential commands that could be issued at runtime.

The widespread use of string-based queries raises security concerns. In particular, this type of query construction requires certain best practices to avoid the introduction of SQLI vulnerabilities. Examples of these practices include using prepared statements and rigorously filtering untrusted inputs before they are used to build a raw query. These security concerns motivate our next two RQs. The first of these, **RQ 4: How do developers use parameterized queries?**, looked at the prevalence and practice of how developers use parameterized queries. A parameterized query leaves certain values unspecified, called parameters or placeholders (labeled with "?"). The actual values of those parameters are bound at runtime. By parameterizing the queries, the database can separate SQL commands from data. This is a common approach to prevent SQLI attacks, as user inputs are syntactically bound to the positions of string literals and are therefore not interpreted as possibly injected commands. The second RQ is **RQ 5: Do developers use tainted inputs to build their raw queries?**. In this RQ, we focused on identifying the sources of data that were being used to construct the string-based database commands. Input sources that derive from untrusted sources can be input vectors for carrying out SQLIs. In this RQ we were interested in quantifying how often this type of data was used to build string-based database commands. Usage of this practice indicates that an app may be vulnerable to SQLI attacks.

Our last set of RQs investigates the energy and runtime performance of the various database commands. We begin our investigation with a high level RQ: **RQ 6: Which database commands are significant in terms of their energy consumption and runtime?**. For this RQ, we quantified the energy and runtime of the various database commands. The results of this investigation showed that the performance of database commands could vary significantly depending on how they were used in the app code. Our next two RQs investigate the context in which the different commands are used. In particular, the first of these two RQs is **RQ 7: Which APIs are most frequently used in loops?**. Loops are a common code construct in mobile apps [9], and intuitively could be combined with database commands to repeatedly perform data related operations. If expensive database related APIs are invoked in loops, their cost can be easily amplified. Therefore, we are interested in knowing which types of database commands most frequently appear in loop–based operations. The results of this RQ revealed a concerning trend — some of the most energy consuming database commands were also the most frequently used in loops. These database commands were those that could modify or change the contents of the database, collectively known as Modification Statement APIs (MSAs). Part of what makes these commands so expensive is that they automatically start a new transaction, if none is in effect, to carry out the command. When these commands are used in a loop, their cumulative cost quickly grows, unless transaction controls are properly used to surround the loop. Our last RQ investigates how often this problematic situation occurred: **RQ 8: Do developers batch database changes made in loops?**.

III. Data Collection

In this section, we describe the design of the experiments for addressing each of the RQs described in Section II. Our general process is as follows: First, we downloaded a large corpus of Android apps from the Google Play app store. We analyzed this set to address some of the high level RQs. We then randomly sampled this set to extract a statistically valid subset that we used for the RQs requiring more detailed analysis. To extract the actual queries from the database APIs, we leveraged an existing string analysis technique, Violist [10], which we give an overview of in Section III-B. For the RQs requiring runtime analysis, we used instrumentation techniques to insert monitoring probes into the apps to assist with recording runtime data while the app was executed. We explain each of these general steps for the data collection in more detail below. Details specific to only one RQ are discussed in the respective text in Section IV.

A. Selection of Subject Applications

We had three criteria for selecting the set of subject applications. We selected apps that: (1) are from different categories – to enable the results to generalize to a broader pool of apps; (2) are top ranked - indicating that the database practices employed in the app, if any, represented a successful usage of database functionality; and (3) could be converted to and from *jimple* bytecode using the *dexpler* [11] tool – since we needed to perform bytecode manipulation of the apps' classes to facilitate measurements and analyses.

To obtain apps that meet the above criteria, we downloaded the top 540 apps in each of the 34 categories defined by the Google Play app store [12] (as of January 2017). Since some apps were incompatible with our device and not all categories had 540 apps in the list of top apps, our *app pool (AP)* contained a total of 12,506 distinct apps. For some of the RQs that require complex analysis, we identified a random subset of 1,000 apps from the AP to serve as an *app sample (AS)*. The AS contained apps from all 34 of the app categories and gave our results a 99% confidence level and a 4% confidence interval when sampling apps from the AP. This ensured a high degree of confidence that our analysis results would be indicative of the larger AP.

B. String Analysis to Identify Queries

Many of the RQs require us to identify the database commands that would be executed by each invocation of a database related API. For RQAs, i.e., `execSQL`, `rawQuery`, and `compileStatement`, the type of command executed is defined in a string parameter that contains plain-text SQL commands. Identifying the possible runtime values of these string parameters is a non-trivial challenge because the strings containing the queries can be built using a sequence of string operations (e.g., concatenation) and may vary over different paths of execution, including branches and loops. Therefore, to accurately identify these queries, we employed a recently developed string analysis, Violist [10], which can identify possible values of a string variable at a given program point.

The Violist analysis is composed of two phases. In the first phase, it generates an Intermediate Representation (IR) of string operations for a particular string variable. The IR captures complex data-flow dependencies, context-sensitive call site information, and the string operations applied to the string variable along the various paths leading to the use of a string variable. In the second phase, Violist applies an interpreter to the IR to generate a model of the possible string values the string variable can have at runtime. Each time we found an RQA in the subject apps, we used Violist to analyze the API's string parameter and identify the set of possible database commands that could be issued at that point.

IV. Experiments, Results, and Discussion

A. RQ 1: Which local databases are most widely used in mobile apps?

Approach: To answer this research question, we statically analyzed the apps in the AP to count database types. To do this, we used Soot to analyze each application's bytecode (i.e., excluding support libraries) and checked the target of all invocations against a list of known database API package names. For example, an invocation target prefixed with "android.database.sqlite" denotes usage of the SQLite database. We identified database package names before beginning the analysis by decompiling all of the apps in the AS and making a list of the class folders present in the app. We then manually examined this list for any class folders that included database–related keywords (e.g., *io, db, base, sql, database, data*) and investigated the corresponding API's documentation to determine if it was for a database. We supplemented this list by consulting online lists of Android databases and added their package names to this list as well. In total, we identified 11 databases or database management systems: SQLite, Oracle, Realm, Couchbase, MongoDB, SnappyDB, LevelDB, Waze, InterBase, UltraLite, and UnQLite.

Results: Table I shows the frequency of database types in the AP. The names of the different databases are shown as the column headers and the number of apps using that database are shown below. The last column "None" represents apps that did not have any invocations of database APIs. Due to limitations in the bytecode handling of Soot, 31 apps could not be analyzed and are not counted in any of the columns.

TABLE I: Distribution of database types in the app pool.

Database	SQLite	Oracle	Realm	Couchbase	MongoDB	SnappyDB	LevelDB	Waze	InterBase	UltraLite	UnQLite	None
# apps	7,314	364	166	6	4	4	2	1	0	0	0	5,097

Discussion: Among all apps, over 59% used at least one database in their implementation. This indicates that local databases are widely used in the mobile app marketplace. Multiple databases were also used within an app. We found that in the AP 40.8% had no database embedded, 55.3% had one database used, 3.8% used two databases, six apps used three databases, but none used more than three databases. Invocations to eight of the known databases were found in the AP. From the table, we can see that the most frequently used databases were SQLite (93.1%), Oracle (4.6%) and Realm (2.1%). SQLite is, by far, the most frequently used database, which is unsurprising since it is shipped with the Android SDK (for free) and actively maintained by the Android development team. Due to the prevalence of SQLite, our remaining RQs focus on this database exclusively.

B. *RQ 2: Which APIs are used to interact with the databases?*

Approach: To address this question, we used Soot to analyze each app's bytecode and counted invocations to the different SQLite related APIs. To identify these, we used a process similar to that in RQ1, but counted an invocation's target method, instead of the package name. As explained at the end of RQ1, we only collected information for their SQLite database (i.e., counts for target methods with the package name "android.database.sqlite"). Since all SQLite APIs are part of the same package, this mechanism was sufficient to locate all SQLite related invocations. For each API that was invoked by a subject app, we counted its invocation frequency over all apps.

Results: Our analysis identified 46,745 invocations to a total of 122 unique SQLite APIs among apps in the AS. The top ten most frequently invoked APIs were `execSQL` (20%), `getWritableDatabase` (8%), `rawQuery` (7%), `endTransaction` (7%), `close` (6%), `delete` (6%), `query` (5%), `update` (4%), `getReadableDatabase` (4%), `insert` (3%).

Discussion: These results show the most common ways of interacting with the database are: accessing the database instance, closing the database, performing transaction control, executing raw queries, and performing database read or write operations. By themselves, this result is somewhat unsurprising since these operations reflect the most basic operations that need to be carried out on a database. However, these results also indicate that they are likely to produce an incomplete picture of how developers are interacting with the database. The reason for this is that almost 30% of the most common interactions are via RQAs (e.g., `execSQL` and `rawQuery`). This means that the actual commands issued by these invocations are defined by their string parameter, which cannot be detected by the straightforward invocation counting approach used in this RQ. This finding motivates the more advanced analysis that we introduce in the next RQ.

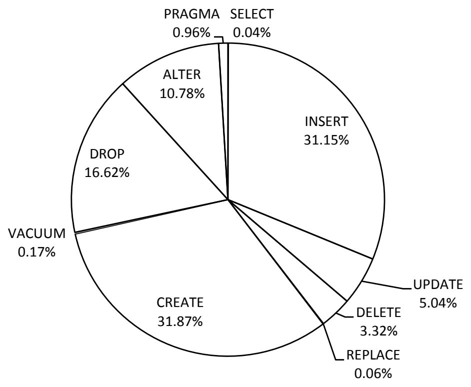

Fig. 1: Distribution of types of queries executed by `execSQL`.

C. *RQ 3: What types of commands are issued as raw queries?*

Approach: To address this RQ, we analyzed the bytecode of each app in the AS to find all invocations to RQAs, and then used Violist to identify the possible database commands they could issue. For each RQA invocation, we ran Violist on the string argument that was used to provide the string-based query. The set of string values returned by Violist represented the possible database commands that could be issued by the invocation. To identify the type of each database command, we analyzed the prefix of each string and matched this against SQL keywords (e.g., "select" or "insert"). This analysis successfully identified the command type for 92% of the strings; the remaining 8% did not begin with a valid prefix and were likely spurious values identified by the analysis along infeasible paths.

Results: We report on the distribution for each of the three possible types of RQAs. For the API `rawQuery`, 99% of the commands issued were `SELECT` queries. The API `execSQL` was used by developers to execute many types of queries. This distribution is shown in Figure 1. The percentages in the figure represent the relative frequency of each possible query type issued by `execSQL`. The API `compileStatement` was mainly used for four types of queries: INSERT (42%), SELECT (26%), DELETE (15%) and UPDATE (14%).

Our analysis also identified 141 instances across 44 apps in which the string values represented multiple queries. These could be identified by the use of the ";" character, which is used by SQL to separate distinct queries in the same string. Almost all of these were arguments to the `execSQL` method, except for two that were executed by the `rawQuery` method. Among these 141 instances of multiple queries, 73% of them were composed of two queries where the second query was either a "COMMIT" or an "END" command. The remaining 27% did not show a discernible pattern.

After we matched the RQA invocations to their corresponding types of commands, we recalculated the invocation frequency of different types of commands issued by developers. The new ranking shows that the frequently issued types of commands in descending order are SELECT, INSERT, `getWritableDatabase`, `endTransaction`, CREATE, `close`, DELETE, UPDATE, `getReadableDatabase` and DROP.

Discussion:

We found that the overall trend we identified in Section IV-B changed significantly after we identified the commands issued by the RQAs. From the results we got in Section IV-B, we could only identify that the common types of queries were DELETE, SELECT, UPDATE, INSERT by looking at the signature of the APIs. However, after studying the RQAs with string analysis, we had a much complete picture of the common database commands issued by developers. A dramatic change was that SELECT and INSERT queries became the most common types of commands, changing from sixth and tenth to first and second. The CREATE, UPDATE and DELETE queries also rank in the top ten.

The analysis of the RQAs also revealed certain problematic developer practices. The official Android documentation provides a list of best practices and requirements for the use of RQAs [13]. The first of these is to not use `execSQL` for INSERT, UPDATE, REPLACE and DELETE types of queries, since they are more easily susceptible to SQLIs. Instead, developers should use specially designed APIs that include additional checks to help prevent SQLIs. For example, developers can use the API `insert` to perform the INSERT type of queries. The second guideline is to not use `execSQL` for SELECT queries, since it does not return any result. Finally, all RQAs are supposed to execute only a single SQL query, and any queries that follow the semicolon ";" will be ignored.

We found a large number of violations of the first best practice. These can make the app vulnerable to SQLI attacks if external inputs are used to build the query. For 39% of all the queries issued by `execSQL`, a more secure version of the API was available. There could be several reasons for the prevalence of this type of violation. One reason may be that in some cases, using `execSQL` is more convenient. For example, in the app PlanningCenter, the statement *INSERT into saved_logins SELECT * FROM temp* is executed by the API `execSQL`. The secure version requires much more programming work to carry out the same action. Another possible reason for the use of the less secure construct is that the developers may have already determined that it contained no external inputs in the query and was therefore, not vulnerable to SQLIs. Although plausible, our initial analysis of the violations indicated that both secure and insecure usages were present. Therefore, this motivated the deeper investigations in RQs 4 and 5.

There were very few violations of the second best practice. Only five instances (0.02%) of all commands issued by the RQAs used `execSQL` for SELECT queries. This suggests that developers generally do not misuse `execSQL` in this way or that the misuse is easily detected by testing practices.

We found some violations of the third best practice in a number of apps. Violating this guideline may or may not introduce failures. In our analysis, we found that for 73% detected instances, the additional query's semantics could be safely discarded. For example, a COMMIT command after an INSERT command did not change the app's behavior. However, we also found some cases where important queries would not be executed. For example, the app `forp` tries to execute forty INSERT queries in one invocation of an RQA. Thirty nine of them will not be inserted into the database. The reason for the prevalence of such violations may be that the mechanism of Android SQLite differs from many other types of databases in handling multiple SQL queries. While many popular database systems support issuing a batch of queries, Android SQLite does not. This finding motivates future research to develop a technique that can detect and repair this sort of violation.

D. RQ 4: How do developers use parameterized queries?

Approach: In this research question, we investigated whether developers parameterized the raw queries issued by the RQAs. A parameterized query leaves certain data values unspecified, called parameters or placeholders. The concrete values of those parameters are bound separately. Generally, all of the RQAs support parameterized queries except for one overloaded version of `execSQL`. Parameterization is a common approach to prevent SQLI attacks. Additionally, using parameterized queries with the API `compileStatement` can provide performance gains because this API will precompile queries. If a precompiled query is later executed repeatedly, the overhead of compiling the query occurs only once.

To identify if the raw queries issued by the RQAs were parameterized, we used the Java SQL parser [14] to parse each of the string values identified by Violist and determined if any placeholders had been embedded in the query string. Note that the use of placeholders is a necessary and sufficient condition for a query to be parameterized. For the parameterized queries, we studied their distribution over the different types of RQAs. For the unparameterized queries, we further investigated if there was any need for parameterization, i.e., if the queries were made up of completely hard coded strings or if dynamic data values were embedded in the queries.

Hard coded strings do not contain any external or dynamic inputs, therefore parameterizing them is not likely to have any security or performance benefits. To identify if a string was hard coded, we analyzed the IR generated by Violist for the particular RQA. Since the IR captures the string operations performed on a string variable, if there was no string operation present in a variable's IR, we could infer that the string-based query was completely defined as a hard coded string.

Dynamic data values that are embedded in a raw query are candidates for parameterization. Using the parser, we identified the substrings that were at the syntactical positions of data values for the queries. If the values of the substrings could not

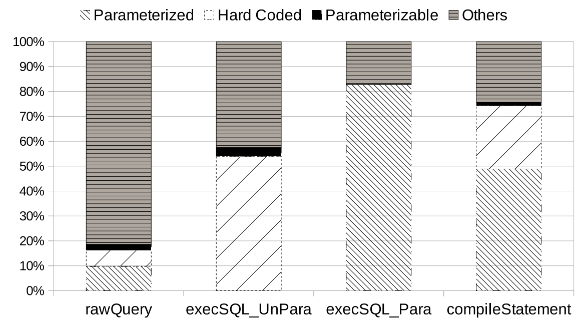

Fig. 2: Distribution of parameterized/unparameterized queries.

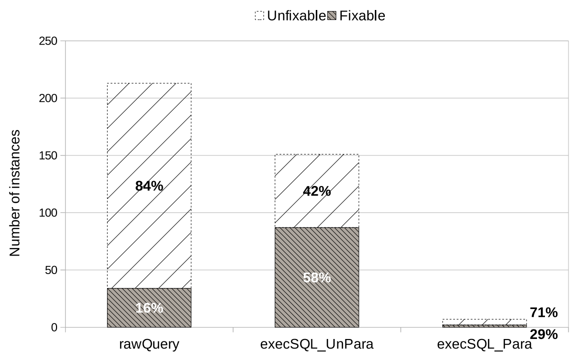

Fig. 3: Distribution of tainted queries.

be resolved statically by Violist, there was a potential need for parameterization (we call such type of unparameterized query *parameterizable query*).

Results: The usage of parameterized queries is shown in Figure 2. The different types of RQAs are shown along the X axis. The ratio of parameterized queries to the total number of queries issued by each type of RQAs is shown along the Y axis (labeled as Parameterized). For `execSQL`, there are two overloaded APIs (denoted as `execSQL_UnPara` and `execSQL_Para`). Since `execSQL_Para` supports parameterized queries but `execSQL_UnPara` does not, we show the ratios for them separately. The unparameterized queries are also shown in Figure 2. The ratio of hard coded unparameterized queries is labeled as "Hard Coded" and the ratio of parameterizable queries is labeled as "Parameterizable". For queries where the substrings at the syntactical positions of data values were constants or the parser could not identify the position of data values, we labeled as "Others".

Our analysis identified 905 instances of parameterizable queries. They appeared in 96 apps (16% of the apps in the AS that used SQLite). The ratio of parameterizable queries by types was SELECT, 3%; INSERT, 7%; UPDATE, 33%; DELETE, 1%; and REPLACE, 0%.

Discussion: Figure 2 shows that the ratios of parameterized queries vary based on the API. The ratios tend to be high for the API `execSQL_Para` and `compileStatement`. This is unsurprising and the reason is likely that these two APIs are specially tailored for parameterized queries. The results also show a non-zero amount of parameterized queries issued by `rawQuery`. Since this API is not optimized for parameterized queries it represents a less than optimal implementation choice by app developers.

The ratios of hard coded unparameterized queries also vary based on the APIs. We can see from Figure 2 that more than half of the unparameterized queries issued by `exec-SQL_UnPara` are hard coded, which is the highest among all of the RQAs. Interestingly, some hard coded unparameterized queries are issued by `compileStatement`. While there is no security benefit for doing this, the developer may by trying to optimize app performance by using the one-time precompiling capability of the API.

As for the parameterizable queries, we can see from the results that there are potentially many apps that can improve their usage of the string-based raw queries. The UPDATE queries have a substantially higher ratio of needing parameterization than the other types. Overall, these findings suggest that there is a need for a technique that can help developers to automatically parameterize their raw queries.

E. RQ 5: Do developers use tainted inputs to build their raw queries?

Approach: To answer this research question, we analyzed the possible queries issued by the RQA invocations and identified the presence of tainted data in the queries. Sources of tainted data are locations in the program where data is read from a potentially untrusted source. To identify these sources, we used the list provided by Susi [15], which contains a list of tainted APIs whose returned values are considered to be untrusted. To identify if a tainted source was used to build a raw query, we used Violist. Since Violist captures the data flow of variables via its IR, any data that derives from a tainted API will be marked as such in the Violist interpreted string value. Therefore, it is straightforward to identify the presence of tainted data in the string-based queries. We call a query that contains untrusted data a *tainted query*. Similar as in Section IV-D, we parsed the tainted queries and investigated if the tainted data was parameterizable.

Results: Our analysis identified 491 tainted queries. They appeared in 50 apps (9% of the apps in the AS that use SQLite). The distribution of tainted queries by APIs was `raw-Query`, 57%; `execSQL_UnPara`, 41%; `execSQL_Para`, 2%; and `compileStatement`, 0%. The detailed distribution is shown in Figure 3. The different types of RQAs are shown along the X axis (`compileStatement` is not included because no tainted queries were executed by this API). The number of tainted queries are shown along the Y axis. The tainted queries that are fixable by parameterization are labeled as "Fixable".

Discussion: We found that developers concatenated tainted inputs to the raw queries in a large number of apps, which could make them vulnerable to SQLI attacks. Interestingly, developers rarely execute tainted queries in the two APIs that are designed for parameterized queries (`execSQL_Para`

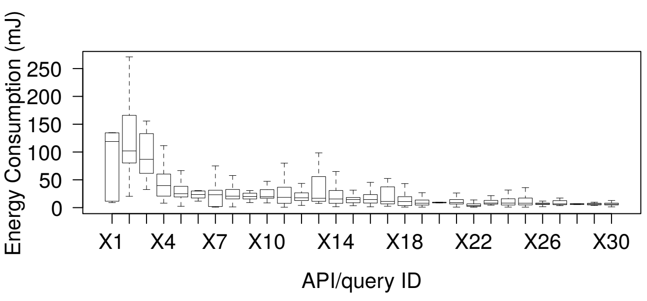

Fig. 4: The median energy usage (in mJ) of the top 30 APIs/queries.

and `compileStatement`). We also found that 25% of the detected instances are fixable by parameterizing the tainted data in query. Overall, these findings motivate future research to help developers make safer use of the RQAs, such as detecting tainted queries and refactoring the code so as to parameterize the queries and harden against SQLI attacks.

F. RQ 6: Which database commands are significant in terms of their energy consumption and runtime?

Approach: To measure the energy consumption and runtime of the database commands, we used the following process: We instrumented each of the apps in the AS with probes to record the start and end times of each invocation to a database API. Then we deployed each app on a Samsung Galaxy S5 that was connected to a power meter. We ran a custom workload on each app and recorded the energy and runtime measurements during the workload's execution.

1) Instrumentation of the Apps: We used instrumentation techniques to rewrite the bytecode of the mobile apps so that the time of execution of each database command would be recorded at runtime. First, using Dexpler, we obtained the *Jimple* representation of the Dalvik bytecode of each subject app. We then used Soot to analyze the app's Jimple code and surround each invocation of a database API with probes that recorded the start and end time of the API invocation. Along with the timing probes, we also inserted instrumentation to record the ID of the thread that executed the invocation, the invocation's signature, and arguments provided to the invocation. This information was recorded outside of the time probes to ensure that it did not affect the time and energy measurements.

2) Deployment: To measure energy, we built a power measurement platform based on the Monsoon power meter [16]. This platform sampled the power consumption of the smartphone at a frequency of 5KHz and synchronized these samples with the standard Unix time. By aligning the execution time of the database commands with the Monsoon measurement samples, we were able to obtain power measurements for each API invocation. We calculated the energy consumption of each API with the function $P * \Delta t$, where P was the measured power of the smartphone while the target API was executed and Δt was its execution time, as measured by the nanosecond level timestamps.

3) Generation of the Subjects' Workloads: To generate the workloads, we manually interacted with each app. We did not use automated crawlers (e.g., PUMA [17]) for the generation of the subjects' workloads since (1) game apps typically required complex operations that could not be generated automatically; (2) a large number of apps required user registration when launched; (3) triggering certain database commands within the app typically required user inputs that could not be easily generated by automated tools; and (4) automated tools often got stuck in some activities due to exceptions. The goal for each workload we created was to be as complete as possible with respect to the app's primary functions. We interacted with each app for 30 seconds to five minutes, depending on the functional complexity of the app.

4) Analysis of Runtime Data: After collecting the runtime data using the above-described process, we calculated the energy consumption of the database commands. The energy consumption of a command was calculated as the sum of all energy consumed between the beginning and ending timestamps. We considered two cases that could make the results inaccurate. In the first case, for APIs executed by concurrent threads, it was not possible to accurately isolate the energy consumed by each of the invocations, since their executions had overlapping time periods. We avoided the mean allocation strategy proposed in related work [18], [8] since it can be inaccurate when invocations in different threads consume significantly different amounts of energy. Hence, we only reported the energy consumption of APIs (including RQAs) that were executed non-concurrently. Their executions accounted for 65.5% of all API executions. In the second case, commands that are included in a `transaction` are bundled into a batch to be executed at the end of the `transaction`. As a result, the energy consumption calculated for these APIs would seem smaller than its actual value. Therefore, we excluded measurements for invocations that were batched with other invocations in the same `transaction`.

Results: We collected measurements for a total of 558,045 executions of 76 unique APIs. Figure 4 shows the results of the energy usage for the top 30 APIs/queries with statistical outliers removed. The x-axis represents the API/query ID, and the y-axis is the corresponding energy usage. Query IDs 1-3, 12, 18 were database initialization commands; IDs 4-8, 11, 13-14, 16, 19, 21-23, and 25 were database write operations; ID 26 was a database read operation; and most of the other IDs were related to setting or getting database related information (e.g., `setVersion`).

Discussion: From the results in Figure 4, we can see that (1) the APIs related to database initialization consume the most energy; (2) database write operations typically consume more energy than database reads; and (3) accessing database related information is the least energy consuming operation. Hence, developers should utilize system caching mechanisms or global initialization to avoid frequently calling operations related to opening or creating a database. Our results also indicate that optimizing APIs related to database write operations could achieve significant energy saving. For the energy usage of string based database commands, six out of all ten commands that were collected are ranked among the top 30 in terms of the energy usage. In Figure 4, their IDs are: 1 (VACUUM), 15 (ALTER), 16 (REPLACE), 19 (DELETE), 22 (UPDATE), and 24 (CREATE). All these commands originate from RQAs. We note that VACUUM consumes much more energy than other queries. This type of query identifies the space that is occupied by deleted rows or other related operations and catalogs it for future use. This process of reclaiming space is costly since it needs to rebuild the database master files.

We further investigated whether the runtime of each of the database commands correlates with its energy consumption. To do this, we applied the Pearson correlation test to the series of time and energy measurements. We found a strong correlation with statistical significance (p-value $< 2.2e\text{-}16$) between energy consumption and runtime. This indicates that for database operations, runtime and energy consumption are strongly correlated. For developers, this means that using techniques to minimize the runtime and workload of database queries is likely to also reduce the energy consumption of their database related operations.

G. RQ 7: Which APIs are most frequently used in loops?

Approach: To address this RQ, we used static analysis of the apps' bytecode to identify invocations that were made in loops for each AS. To identify loops, we implemented a region based analysis, where each region corresponds to a loop or method body [19]. We then analyzed each region to determine which, if any, invocations to database APIs were contained in the loop bodies. We then ranked the target APIs according to their invocation frequency.

Results: The most frequent APIs invoked in loops are `execSQL`, `rawQuery`, `getWritableDatabase`, the constructor of `SQLiteOpenHelper`, `isOpen`, `insert`, `delete`, `query`, `update`, and `getReadableDatabase`.

Discussion: The results show that the most frequent APIs invoked in loops are similar to the top ones in the overall frequency ranking shown in Section IV-B. In our analysis, we found that several high energy consuming APIs appear frequently in loops and can be repeatedly invoked. For example, the API `openDatabase`, which is in the 79th percentile in terms of energy consumption, is frequently used to access an existing database file in a loop. In this situation, if the app accesses the same database in every loop iteration, the API invocation could possibly be moved out of the loop to save energy and improve performance. Several energy expensive APIs, `execSQL`, `insert`, `update`, and `delete`, also appear frequently in loops. If these particular invocations are not batched properly, they may consume a high amount of energy. Therefore, in the next research question, we analyze the use of transactions with respect to these APIs.

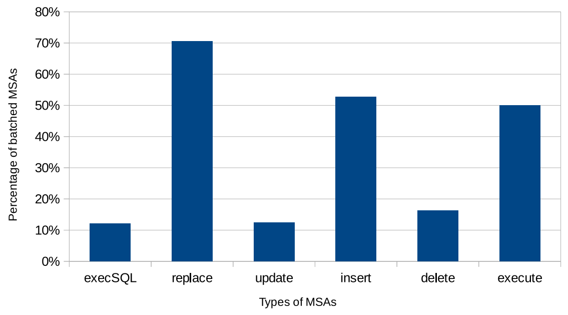

Fig. 5: Percentage of batched MSAs over all of the MSAs in loops.

H. RQ 8: Do developers batch database changes made in loops?

Approach: Database changes made in a loop can be very expensive in terms of both energy consumption and runtime (Section IV-F). In Android SQLite, MSAs are a set of APIs that are responsible for changing the contents of the database. Part of what makes them so expensive is that they automatically start a new transaction (called an autocommit), if none is in effect, to carry out the command. Transactions are expensive because they require file locking and rollback capability. Therefore, if developers use a single transaction to batch MSAs inside of a loop, it could achieve significant energy savings. This can be done by starting an explicit transaction (i.e., `beginTransaction`) before the loop starts and ending the transaction (i.e., `endTransaction`) after the loop ends. To identify if database changes made in loops were batched or not, we developed a static transaction analysis. There are two phases in the analysis. First, it identifies the statements that invoke MSAs inside loops by analyzing the Control Flow Graphs (CFGs) of the app. Second, for each identified statement, the analysis models the transaction open and close operations along the control flow paths that could lead to the statement. If each explicit transaction opened has been closed when the statement executes, it means the database change made by the statement in the loop is not batched. We ran the transaction analysis on each of the apps in the AS. For ease of explanation, we call the MSAs that are invoked in loops and batched by developers *batched MSAs* (similar for *unbatched MSAs*).

1) Batched MSAs: **Result:** The analysis identified 103 apps in which there exists at least one batched MSA. These 103 apps represent 38% of the 266 apps that contain MSAs in loops. On average, 21% of the instances of the MSAs invoked in loops are batched. Figure 5 shows the percentages of different types of batched MSAs. The items displayed on the X-axis are the different MSAs.

Discussion: The APIs `execSQL`, `update` and `delete` are batched less often than the other MSAs. They have batch rates at around 10% to 15% while `insert` and `replace` and `execute` have batch rates over 50%. As we show in Section IV-F, `execSQL`, `update` and `delete` can be very expensive in terms of energy and runtime. They also appear frequently in loop operations (see Section IV-G). These results seem to indicate that for some database commands, developers are much better at including them in transactions than others. The reason for this is unclear from the results, but could be due to a lack of developer awareness of the underlying auto-commit behaviors for the different commands.

2) Unbatched MSAs: **Result:** The transaction analysis identified 220 apps in which there exist at least one unbatched MSAs. These 220 apps represent 82% of the 266 apps that contain MSAs in loops and 37% of the 583 apps that contain SQLite usage. The average number of unbatched MSAs per app is 9 with a median of 4.

Discussion: The results show that there are many potential opportunities to improve the apps by optimizing the unbatched MSAs. To better understand these unbatched MSAs, we collected statistics about the unbatched MSAs. First, 28% of the unbatched MSAs are invoked in the main thread. In Android, there is a single thread (called main thread) that is responsible for rendering the UI. Unbatched MSAs are usually expensive in terms of energy and runtime. Executing them in the main thread can directly affect the responsiveness of the UI.

Although there are many opportunities to optimize MSAs and the optimization can be very beneficial, carrying out the optimization can be challenging. Since the batching requires developers to use the same database object that invokes the MSA to place a `beginTransaction` call before the loop starts, if the call chain between the loop and the MSA is complex, it may be challenging to locate the correct object. We also examined the call path length between the method that contains the MSA and the method that contains the loop. If the length is zero, it means they are in the same method. Our analysis detected that 66% of the unbatched MSAs are inter-procedural (i.e., call path length is larger than zero). Moreover, 22% of the unbatched MSAs have a call path length between two and seven. These numbers indicate that batching the MSAs requires tracing a complex call chain and suggests that such optimizations may be challenging for developers without automated support.

Other than the complex call chains, another challenge for optimization is that the database object may not be available before the loop starts, i.e., it is defined inside the loop. Our analysis detected that 31% of the unbatched MSAs have the database object defined inside the loop. This means it may not be available/initialized before the loop starts. Interestingly, if we combine this information with the call path length between the MSA invocation and the loop header, we found that when the call path lengths go from zero to seven, the percentages of the unbatched MSAs that have the database object defined in the loop are 2%, 32%, 61%, 63%, 45%, 45%, 100% and 100%, respectively. For these unbatched MSAs, the optimization may need to consider moving the definition of the database object out of the loop so as to be able to invoke `beginTransaction` outside of the loop.

Overall, our findings suggest that there are many opportunities to optimize the apps in terms of unbatched MSAs and the optimization may even improve the responsiveness of the UI. However, developers may need a technique that can help them detect and optimize the unbatched MSAs due to the complexity of the unbatched MSAs.

V. Artifact Description

To facilitate research into local database usage in mobile applications, we have made available the raw data and analysis results that were used in our paper. Our artifact was reviewed and approved by the ICSME 2017 Artifacts Committee and includes the following material:

- The statistics of app-level local database usage (RQ1).
- The detailed static analysis results of the API analysis (RQ2), raw query related analyses (RQ3, RQ4, RQ5), loop analysis (RQ7), and transaction analysis (RQ8).
- The energy consumption measurements of SQLite related APIs (RQ6), as well as the energy raw data and runtime usage log.
- The analysis code and scripts that were used to produce the results.

Note that due to app distribution regulations, our artifact does not contain the apps used in our evaluation, only the analysis results. Our artifact is available under the Apache Software License v2.0 from https://github.com/USC-SQL/SQLUsage.

VI. Threats to Validity

External Validity: A potential threat is that our study did not include paid versions of apps, which may be tested and designed differently than free apps. Therefore, our conclusions may not apply to paid apps. However, it is important to note that our results are still representative of most Android apps, since almost 70% of all Android apps are free apps [20].

Most of our analysis focused on SQLite, which could have different usage patterns than other database types. This would mean our results may not generalize beyond SQLite. However, SQLite dominates overall database usage and its mechanisms are similar to many other database types that appeared in our study. Therefore, we argue our results are meaningful to most developers and most apps.

For several of the RQs, we used the AS instead of the AP. The reason for this is that Violist is a complex and time-consuming analysis. To ensure that our results could generalize more broadly to the AP, we chose a subset size that gave us a 99% confidence level and a 4% confidence interval when sampling apps from the AP. Furthermore, the AS contained apps from all 34 categories. As a further check, we compared the database type distribution in the AS and AP, and found that AS had a highly similar distribution as the AP. This provided us with a high degree of confidence that our analysis results were indicative of the larger AP.

The energy and runtime measurements were conducted on a single device, a Samsung Galaxy S5 running Android 5.0. Although we believe this phone and operating system to be representative of most smartphones, if other devices and operating systems have significantly different energy consumption or runtime characteristics, then our results may not hold for those platforms. In particular, our results for RQ6 would need to be reevaluated.

Internal Validity The energy consumption of APIs/queries may be impacted by different runtime contexts, and by whether all database related functions were executed during the workload generation. However, this potential bias was mitigated by the large number of applications and API executions in our experiment.

We identified 11 different and widely used Android local databases based on online reports (Section IV-A), but it is possible that our list is incomplete. This would only impact our results from RQ1, since the method of identifying APIs for the other RQs required an invocation to explicitly be part of the SQLite package.

We used automated analysis tools to identify and classify characteristics of our subjects. Inaccurate analysis could impact the validity of our results. However, these tools have been shown to have high accuracy, which minimizes this threat. A previous evaluation of Violist showed that it had an average precision of 86% and a recall of 100% on marketplace apps. We conducted a study on the transaction analysis tool (used in RQ8) and found it had a precision of 100%. Although the recall rate of this tool is unknown, false negatives would not undermine our claims (e.g., the prevalence of optimizable apps) since the precision provides a lower bound on the results.

VII. Related Work

Some existing work that focuses on the energy consumption of mobile applications also reports database energy usage. Linares-Vásquez and colleagues [8] conducted an empirical study on measuring the energy consumption of Android API methods. In their study, they mined the usage patterns of energy greedy APIs. Our study is different in that we employed and adapted a range of dynamic and static program analysis techniques to discover different and more detailed patterns of database API usages (e.g., raw SQL queries and transactions), which cover not only API level energy information, but also command level energy consumption, performance, and security aspects of the apps. Li and colleagues [7] investigated the energy consumption of over 400 real-world marketplace mobile applications. Their conclusion that SQLite is one of the most energy consuming components in Android apps inspired us to conduct the more detailed analysis presented in this paper. Eprof [21] studied the energy usage of certain APIs and showed that I/O is one of the major energy consuming components. However, the energy consumption of database related APIs was not addressed.

The security aspects of database usage in mobile applications have been studied in related work. Various tools, such as AppSealer [4], IntentDroid [5], and CHEX [6], were proposed to detect or prevent security vulnerabilities (e.g., component hijacking, inter-application communication) of Android apps. Zhou and colleague [22] assessed the prevalence of two vulnerabilities in Android apps at a large scale. All of the above work considered local databases as one of the vulnerable targets, but did not provide a detailed picture of database usage in Android apps nor evaluate the prevalence of certain problematic patterns of query construction. We believe our results are complementary to this body of related work, and help developers to better understand app vulnerabilities.

Another group of related work analyzed database usage among applications from different perspectives. Qiu and colleagues [23] conducted an empirical analysis of the co-evolution of database schemas and code in ten large database applications. They found database schemas evolved frequently during the application life cycle and caused significant code-level modifications. Goeminne and colleagues [24] explored the survival of five Java database frameworks among different Java projects to see whether certain database frameworks were used more successfully in certain combinations than others. Decan and colleagues [25] empirically studied how the use of relational database access technologies evolved over time in open source Java projects. In contrast, we focused on the local database usage in mobile apps and provided a detailed look into the associated practices, costs, and potential problems. Meurice and colleagues [26], [27] presented a static analysis approach to understand dynamic database usage in Java systems. There are two major differences between their study and ours. First, their work focused on identifying the table and column names of interpreted queries. Our work analyzed different characteristics of the queries, such as type, parameterizability, and tainted information. Second, the string analysis used in their work is limited in its ability to deal with complex string manipulation and data flow. This can result in false positives and negatives in identifying string values.

Some existing work investigates or detects performance issues in mobile apps. The performance issues can be caused by implementation choices [28], performance bugs [29], [30], code smells [31] and mobile ads [2]. One way to detect these issues is to manually diagnose the recorded resource usage information [32], while static analyses [29], [30], [33] can be used to automatically detect some performance bugs. Nevertheless, none of above work focuses on studying the performance impact of local database usage on mobile apps.

Previous studies of mobile apps have also provided tools to model or optimize app performance. For example, Mantis [34] predicts the execution time of Android apps on given inputs. Different models [18], [21], [35], [36], [37], [38], [39], [40] have also been proposed to estimate the energy consumption of various aspects of mobile apps. Other techniques focus on detecting specific performance problems or bugs, such as display energy [41], [42], sensory data utilization [43], concurrency [44] and UI performance [45]. Some researchers have also proposed techniques to optimize the app performance by minimizing display energy [46], [47], [48], [49] or network request energy [50]. However, the resource costs of database usage in mobile apps were not investigated in these studies.

VIII. Conclusions

In this paper, we presented the results of an extensive empirical study on local database related behaviors in mobile apps. Our study revealed several interesting observations. We found a large number of violations of many types of best practices regarding database usage in mobile applications. From a security perspective, we found that developers frequently use vulnerable APIs and patterns to execute and build SQL commands. A large number of these instances were avoidable and could be refactored to prevent SQLI attacks by using secure APIs or parameterized queries. From the energy/performance perspective, we found that database initialization and write operations are the most expensive. We also found that these operations appear frequently in loop structures, many of which are not properly batched in explicit transactions and can cause significant inefficiencies. Overall, our study provides interesting insights that give concrete guidelines for app developers to improve their apps and motivate areas for future research work in the software engineering community.

IX. Acknowledgments

This work was supported, in part, by the National Science Foundation under grant numbers CCF-1321141 and CCF-1619455.

References

[1] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, “What Do Mobile App Users Complain About?” *IEEE Software*, vol. 32, no. 3, pp. 70–77, May 2015.

[2] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond, “Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. IEEE Press, 2015, pp. 100–110.

[3] J. Gui, M. Nagappan, and W. G. Halfond, “What Aspects of Mobile Ads Do Users Care About? An Empirical Study of Mobile In-app Ad Reviews,” *arXiv preprint arXiv:1702.07681*, 2017.

[4] M. Zhang and H. Yin, “AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications,” in *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS’14)*, San Diego, CA, February 2014.

[5] R. Hay, O. Tripp, and M. Pistoia, “Dynamic Detection of Inter-application Communication Vulnerabilities in Android,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. ACM, 2015, pp. 118–128.

[6] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12. ACM, 2012, pp. 229–240.

[7] D. Li, S. Hao, J. Gui, and W. G. J. Halfond, “An Empirical Study of the Energy Consumption of Android Applications,” in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2014, pp. 121–130.

[8] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, “Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. ACM, 2014, pp. 2–11.

[9] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, “CLAPP: Characterizing Loops in Android Applications,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 687–697.

[10] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, “String Analysis for Java and Android Applications,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 661–672.

[11] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, “Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot,” in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, ser. SOAP ’12. ACM, 2012, pp. 27–38.

[12] “https://play.google.com/store/apps.”

[13] Google, “Android SQLite Documentation,” https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html, 2017.

[14] Tobias, “JSQLParser,” https://github.com/JSQLParser/JSqlParser, 2017.

[15] S. Rasthofer, S. Arzt, E. Spride, T. U. Darmstadt, and E. Bodden, “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks,” in *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS’14)*, San Diego, CA, February 2014.

[16] Monsoon Solutions, Inc, “Monsoon Power Monitor,” http://www.msoon.com/LabEquipment/PowerMonitor, 2017.

[17] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, “PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps,” in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’14. ACM, 2014, pp. 204–217.

[18] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, “Calculating Source Line Level Energy Information for Android Applications,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. ACM, 2013, pp. 78–89.

[19] S. S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[20] Statista, “Distribution of Free and Paid Android Apps in the Google Play Store from 2009 to 2015,” https://www.statista.com/statistics/266211/distribution-of-free-and-paid-android-apps/, 2017.

[21] A. Pathak, Y. C. Hu, and M. Zhang, “Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys ’12. ACM, 2012, pp. 29–42.

[22] Y. Zhou and X. Jiang, “Detecting Passive Content Leaks and Pollution in Android Applications,” in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS’13)*, San Diego, CA, February 2013.

[23] D. Qiu, B. Li, and Z. Su, “An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. ACM, 2013, pp. 125–135.

[24] M. Goeminne and T. Mens, “Towards a Survival Analysis of Database Framework Usage in Java Projects,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 551–555.

[25] A. Decan, M. Goeminne, and T. Mens, “On the Interaction of Relational Database Access Technologies in Open Source Java Projects,” *arXiv preprint arXiv:1701.00416*, 2017.

[26] L. Meurice, C. Nagy, and A. Cleve, *Static Analysis of Dynamic Database Usage in Java Systems*. Cham: Springer International Publishing, 2016, pp. 491–506.

[27] L. Meurice and A. Cleve, “DAHLIA 2.0: A Visual Analyzer of Database Usage in Dynamic and Heterogeneous Systems,” in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, Oct 2016, pp. 76–80.

[28] L. Corral, A. Sillitti, and G. Succi, “Mobile Multiplatform Development: An Experiment for Performance Analysis,” *Procedia Computer Science*, vol. 10, pp. 736 – 743, 2012.

[29] Y. Liu, C. Xu, and S.-C. Cheung, “Characterizing and Detecting Performance Bugs for Smartphone Applications,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 1013–1024.

[30] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, “What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps,” in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’12. ACM, 2012, pp. 267–280.

[31] G. Hecht, N. Moha, and R. Rouvoy, “An Empirical Study of the Performance Impacts of Android Code Smells,” in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft ’16. ACM, 2016, pp. 59–69.

[32] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda, “Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance,” in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sept 2013, pp. 1–10.

[33] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, “Characterizing and Detecting Resource Leaks in Android Applications,” in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE’13. IEEE Press, 2013, pp. 389–398.

[34] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic Performance Prediction for Smartphone Applications," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 297–308.

[35] L. Zhang, B. Tiwana, R. P. Dick, Z. Qian, Z. M. Mao, Z. Wang, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2010, pp. 105–114.

[36] Y. F. Chung, C. Y. Lin, and C. T. King, "ANEPROF: Energy Profiling for Android Java Virtual Machine and Applications," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, Dec 2011, pp. 372–379.

[37] M. Dong and L. Zhong, "Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. ACM, 2011, pp. 335–348.

[38] J. Gui, D. Li, M. Wan, and W. G. J. Halfond, "Lightweight Measurement and Estimation of Mobile Ad Energy Consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, ser. GREENS '16. ACM, 2016, pp. 1–7.

[39] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating Android Applications' CPU Energy Usage via Bytecode Profiling," in *Proceedings of the First International Workshop on Green and Sustainable Software*, ser. GREENS '12. IEEE Press, 2012, pp. 1–7.

[40] ——, "Estimating Mobile Application Energy Consumption Using Program Analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 92–101.

[41] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond, "Detecting Display Energy Hotspots in Android Apps," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, April 2015, pp. 1–10.

[42] M. Wan, Y. Jin, D. Li, J. Gui, S. Mahajan, and W. G. J. Halfond, "Detecting Display Energy Hotspots in Android Apps," *Software Testing, Verification and Reliability*, 2017.

[43] Y. Liu, C. Xu, and S. C. Cheung, "Where Has My Battery Gone? Finding Sensor Related Energy Black Holes in Smartphone Applications," in *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2013, pp. 2–10.

[44] Q. Li, Y. Jiang, T. Gu, C. Xu, J. Ma, X. Ma, and J. Lu, "Effectively manifesting concurrency bugs in android apps," in *2016 23rd Asia-Pacific Software Engineering Conference*, ser. APSEC '16. IEEE Press, 2016, pp. 209–216.

[45] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "DiagDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 410–421.

[46] D. Li, A. H. Tran, and W. G. J. Halfond, "Making Web Applications More Energy Efficient for OLED Smartphones," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 527–538.

[47] ——, "Nyx: A Display Energy Optimizer for Mobile Web Apps," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 958–961.

[48] ——, "Optimizing Display Energy Consumption for Hybrid Android Apps (Invited Talk)," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, ser. DeMobile 2015. ACM, 2015, pp. 35–36.

[49] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 143–154.

[50] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated Energy Optimization of HTTP Requests for Mobile Applications," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 249–260.