CURATOR — A Secure Shared Object Store

Design, Implementation, and Evaluation of a Manageable, Secure, and Performant Data Exchange Mechanism for Smart Devices

Christoph Stach
University of Stuttgart, IPVS / AS
Stuttgart, Germany
stachch@ipvs.uni-stuttgart.de

Bernhard Mitschang
University of Stuttgart, IPVS / AS
Stuttgart, Germany
mitsch@ipvs.uni-stuttgart.de

ABSTRACT

Nowadays, smart devices have become incredibly popular—literally everybody has one. Due to an enormous quantity of versatile apps, these devices positively affect almost every aspect of their users' lives. E. g., there are apps collecting and monitoring health data from a certain domain such as diabetes-related or respiration-related data. However, they cannot display their whole potential since they have only access to their own data and cannot combine it with data from other apps, e. g., in order to create a comprehensive electronic health record. On that account, we introduce a s**ecur**e sh**a**red objec**t** st**or**e called CURATOR. In CURATOR apps cannot only manage their own data in an easy and performant way, but they can also share it with other apps. Since some of the data is confidential, CURATOR has several security features, including authentication, fine-grained access control, and encryption. In this paper, we discuss CURATOR's design and implementation and evaluate its performance.

CCS CONCEPTS

• **Information systems** → **Data exchange**; *Database design and models*; Data encryption; Database performance evaluation; • **Security and privacy** → **Security services**; *Access control.*

KEYWORDS

Data exchange, smart devices, shared object store, security

ACM Reference Format:
Christoph Stach and Bernhard Mitschang. 2018. CURATOR — A Secure Shared Object Store: Design, Implementation, and Evaluation of a Manageable, Secure, and Performant Data Exchange Mechanism for Smart Devices. In *SAC 2018: SAC 2018: Symposium on Applied Computing , April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3167132.3167190

1 INTRODUCTION

Smartphones and similar smart devices have become increasingly popular over the past decade. The reason for this phenomenal approval on behalf of general public is their rich sensing capabilities. While these devices approximate more and more to desktop PCs in terms of computational power, their ability to gather knowledge about the users is what really stands out. This leads to millions of apps being developed and used. These apps are not just toys for technical enthusiasts, but assist their users in various domains, such as healthcare [30] or at their workplace [18].

However, the rise of such apps has not only created many new opportunities and innovative cases of application, but it also has led to novel challenges. Contrary to operating systems for desktop PCs, mobile platforms such as Android or iOS execute apps in strictly isolated sandboxes. iOS applies this application isolation on several execution layers. On the one hand, this makes it almost impossible for malicious apps to intercept data from another app. On the other hand, also a legitimate data exchange between apps is severely limited [19]. Android is one of the few mobile platforms enabling a flexible data exchange between apps to a certain extent. However, the provided data exchange techniques cause a significant programming overhead for app developers [20].

Moreover, the applied data exchange mechanisms are highly insecure [7]. Even Android's private databases pose a threat for the stored data as there are many documented weak spots in the implementation of these databases [14]. Since smart devices hold a lot of sensitive data (e. g., private data such as health data or confidential data such as business data) additional security challenges come along as such data require special protection measures [17]. The security mechanisms applied in Android are insufficient for this purpose [29]. Thus, a data exchange mechanism for smart devices must not only focus on usability but also on data security.

The flash memory used to store data in smart devices is principally very fast. Nevertheless, disk accesses are often the bottleneck in apps [15]. Thus, a data exchange mechanism also has to keep performance issues in mind—especially its impact on the runtime and the battery drain of an app. For data management, Android supports only relational databases (namely *SQLite*[1]) out of the box. However, relational databases are tailored to frequent but small transactions or huge batch processing with rare writing operation. On the contrary, multiple concurrent data requests and frequent data changes are less efficient in such a database system [1]. It has to be clarified whether a relational database is a sound foundation for a data management and data exchange mechanism.

For all of the reasons above, we introduce a novel s**ecur**e sh**a**red objec**t** st**or**e for smart devices called *CURATOR*. CURATOR is a shared data container which can be used by any kind of app. In

[1] see https://www.sqlite.org

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167190

CURATOR apps can store objects and specify for each object individually which other apps may have access to it. Wrapper classes deal with the (de)serialization of the stored objects in order to keep the overhead for developers marginally. All stored data is fully encrypted and an access control system ensures data security. The concept of CURATOR is abstracted from a specific storage. Thereby, the best suited storage technology can be selected.

In this paper, we yield the following five contributions: **(I)** We study data exchange mechanisms applied in today's mobile platforms. **(II)** We introduce a concept for a manageable, secure, and performant data exchange mechanism for smart devices, called CURATOR. **(III)** We analyze the usage of relational databases, key–value stores, and document stores for smart devices. **(IV)** We implement CURATOR using each of the three analyzed database technologies. **(V)** We evaluate CURATOR and identify performance characteristics for each of the three database technologies.

The work at hand focuses on Android and the CURATOR prototypes run on Android. This is due to two reasons: Only Android enables apps to exchange data with apps in other sandboxes. Moreover, Android is by far the prevalent mobile platform according to IDC Research [2] and it also reaches out for the Internet of Things (IoT) [9]. So, the gained insights can be transferred to any mobile or IoT platform.

The remainder of this paper is structured as follows: Section 2 attends to related work. This comprises both, state of the art as well as current research work. Then, we introduce our approach in Section 3. We detail on CURATOR's implementation in Section 4: Section 4.1 discusses the applicability of key-value stores, document stores, as well as relational databases in CURATOR and Section 4.2 describes its technical realization. Finally, we perform a comprehensive evaluation of three different CURATOR implementations in Section 5, before Section 6 concludes this paper.

2 RELATED WORK

In the context of this work, we look at both, Android's built-in data exchange mechanisms and third-party approaches. We mainly consider usability and security aspects. A performance analysis is given as part of the evaluation (Section 5).

Android Data Exchange Mechanisms. The most basic data exchange mechanism supported by Android is the public file system. Files saved on the *External Storage* are available to any app[2]. The data owner, i. e., the app that created the file initially, cannot specify which app should have access to its files or what an app is allowed to do with a file. These files can be read, modified, and deleted unrestrictedly. Android also allocates a private partition on the *Internal Storage* for each app. By default, files saved on the Internal Storage are only available to the data owner. The owner can attach special flags to a file at creation time to indicate that this file should be accessible for any app. The sole advantage of the Internal Storage compared to the External Storage in terms of security is, that the owner specifies whether a file can only be read or also modified by other apps [13]. This approach causes a huge overhead for developers since they have to mind the (de)serialization of the data. Also, each developer has to know the file formats used by other apps, as there is no predefined formatting standard. Android supports full disk encryption (FDE) to secure its file system. Yet, Android's FDE does not operate on the whole storage and this protection is maintained only until the device is unlocked [6].

A data exchange mechanism that allows to pass data from one app to another at runtime are the so-called *Explicit Intents*. The data owner has to call a certain app explicitly which should have access to its data. Then, the affected app is started and it has to process the passed data instantly. The data owner has to indicate a distinct Intent for every app it wants to exchange data with. Also, each called app has to know how to process these Intents. Since only primitive data types and composite data types such as arrays or lists are supported, higher-order data types have to be decomposed apriori [12]. This leads to high efforts for developers. Nevertheless, a data owner is able to restrict which other apps have access to its data. Yet, an encryption of the shared data is not provided. This is aggravated by the fact, that this data exchange mechanism is downright insecure [3].

An approach with a focus on simple usability is the *Clipboard Framework*. Apps can temporary clip data on the Clipboard to make it available to any other app. Any data type is supported. Each object is annotated with its type, in order to facilitate the integration of the data into an app. However, the Clipboard holds only one clip object at a time and subsequently added objects overwrite previous ones [11]. To enable a simplified accessibility, security is neglected. No permissions are required to use the Clipboard, a data owner cannot restrict access to its data, and data encryption is not supported. That is why several malware apps misuse the Clipboard either to steal private data or manipulate the clipped data [40]. Moreover, the lifetime of the clipped data is not linked to the one of its owner, i. e., if an app is uninstalled, its data remain on the Clipboard [41].

The most promising built-in data exchange mechanism are the so-called *Content Providers*. Content Providers constitute interfaces to an app's data. Regardless of the format the provided data is kept internally, the interface provides standardized query, insert, update, and delete functions. The data owner has to implement these functions for each of its Content Provider. In this way, the owner is able to exclude certain data or even obfuscate the data before releasing it via the Content Provider. When another app uses a Content Provider, the data owner is called, performs the actual data access, and forwards the data to the caller. Basically, any app is able to use a Content Provider, but the data owner may specify a custom permission restricting the usage [10]. However, any app is able to request this permission and it is automatically granted at installation time since users are overchallenged by Android's permission mechanism [8]. Apart from this, also the Content Provider mechanism itself is vulnerable towards various attacks [28, 42].

Third-party Data Exchange Mechanisms. *Mobius* [5] introduces an infrastructure that enables to exchange data between both, apps and smart devices. Each app has access to a private virtual database which is realized as a partition of a system-wide physical database. Data can be shared with other apps by transferring it to the respective partitions. The central database is synchronized with a cloud database to make the data available for other devices. The focus of Mobius is on need-based data provisioning and intelligent caching strategies. Access control or data security is not addressed.

[2] A permission to access the External Storage is required.

Moreover, neither privacy nor controlling mechanisms for data location are considered, although all data is transferred to an external resource.

Kynoid [26] is a security mechanism for every Android data source, e. g., Content Providers. The system introduces fine-grained access rights which are audited at runtime. Thereby, a user is able to specify which app is allowed to access what data. Moreover, a spatio-temporal context can be added to the access rights to limit their scope. Yet, Kynoid is only applicable to Android's data exchange mechanisms—thus it also suffers from their weak spots in terms of operability.

MetaService [4] is an enhanced but easy to use Clipboard mechanism. While Android's Clipboard is not able to operate with higher-order data objects and has to decompose each object into its attributes, MetaService directly processes objects. It introduces a meta data management to handle various data types and deal with their casting. However, data security is completely out of scope and the exchange capacity is limited to one object at a time.

Poscha [22] introduces a digital rights management system for Android. Any kind of data can be annotated by context-sensitive policies specifying who (i. e., which app) is allowed to do what (e. g., read or modify) with which data. Poscha controls any kind of data usage and ensures that it complies with the policies. Yet, it provides no data management mechanism and data exchange has to be realized manually.

The *Secure Data Container (SDC)* [36] is a shared database with several security features. For instance, the database is fully encrypted and the SDC introduces an authorization mechanism as well as a fine-grained access management system. Due to this system, a data owner is able to specify for each tuple independently which apps are allowed to access the data. The data is stored in a relational database. By default, it offers a key-value database schema. Yet, this basic schema can be extended (see Figure 2). However, data objects have to be split into attributes in order to store them. This causes an overhead because of the (de)serialization.

SeSQLite [21] extends Android's SQLite implementation by adding security features. It is based on Android's implementation of *SELinux*[3]. That way, fine-grained mandatory access control at both, schema and row level is introduced in SeSQLite. Still, developers have to mind how to share their data with other apps. Moreover, in order to use SeSQLite, the Android system has to be modified manually which is why this approach is not appropriate for common users.

3 CURATOR'S ARCHITECTURE

Since none of the available approaches towards a data exchange mechanism is entirely satisfying in terms of usability, security, and performance, we come up with our own approach called CURATOR. For this purpose, we combine the easy usability of MetaService with the data management and security features of the SDC. I. e., CURATOR is a shared object store which operates with any data type. Opposed to MetaService, CURATOR has no limitation in terms of storage capacity. CURATOR adopts security features from SDC including authentication, authorization, access control, and data encryption. We address the following issues:

[3] see https://source.android.com/security/selinux/

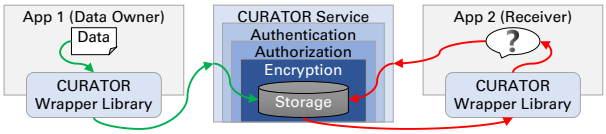


Figure 1: CURATOR's Architecture (Data Storage Process Depicted in Green; Querying Process Depicted in Red).

Simple Usability. The key aspect of CURATOR is its usability. That is, the inclusion of CURATOR into apps has to be easy and data management (including data sharing) has to be facilitated. Necessary to that end is a simplified interface and permanent availability—the latter is achieved by realizing CURATOR as a background service that is called at system startup. The interface includes generic functions for inserting, updating, sharing, and deleting objects. At that, CURATOR supports higher-order data types. An app developer passes an object to CURATOR and the (de)serialization is realized automatically. That way, storage abstraction can be achieved. I. e., the applied database technology and schema remains transparent towards apps.

Data Security. Analogous to the SDC, an app has to authenticate to CURATOR before it is able to use its services. CURATOR has also an authorization mechanism that enables users to specify which app is allowed to use which functions of CURATOR. For each stored object one of four release levels (*private*, *readable*, *updatable*, and *deletable*) is declared. Additionally, a list of apps can be specified for which this release level is applied—by default, all data is private and only the data owner has full access to it. An access control system enforces these rules. All stored data is fully encrypted. That way, the data is secured, even in case of a compromised smart device. By deleting the corresponding key, data can also be deleted reliably. Finally, the scope of the stored data is linked to its owner's lifetime. I. e., data is automatically deleted when its owner is uninstalled.

Performance. Despite of the security features, CURATOR is performant due to its storage abstraction. The best suited database technology can be applied depending on the respective use case (see Section 4.1 for the available technologies).

Figure 1 shows the onion architecture of CURATOR, i. e., it consists of several nested layers. Apps can only interact with the *CURATOR Service Layer* (read, insert, update, or delete). For each function call, the *Authentication Layer* verifies the caller's identity. Then, the *Authorization Layer* checks, whether the call is legitimate. The *Encryption Layer* deals with the (de)ciphering of the corresponding data. The actual operations are executed at the *Storage Layer*.

The data storage process is shown on the left side. An app using CURATOR has to include the *CURATOR Wrapper Library*. A data owner sends its data to this wrapper. The wrapper handles the serialization of the data and forwards it to the CURATOR service. An app using CURATOR for the first time has to register at the service. Subsequently, this registration data is used for authentication. For data storage, CURATOR only has to check, whether an app is allowed to use the service. Then, the data is encrypted and stored. The querying process (right side) also verifies the caller's identity initially. During authorization, CURATOR also has to check whether the caller is the data owner or whether the affected data is shared with the caller explicitly. If the caller has the required

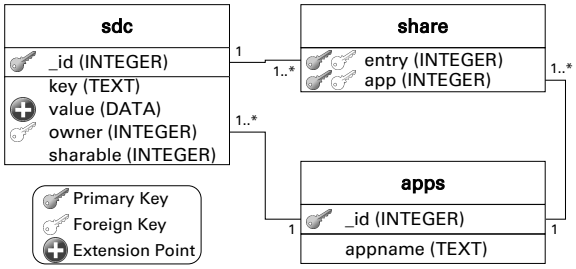


Figure 2: Database Schema Applied in the SDC (cf. [33]).

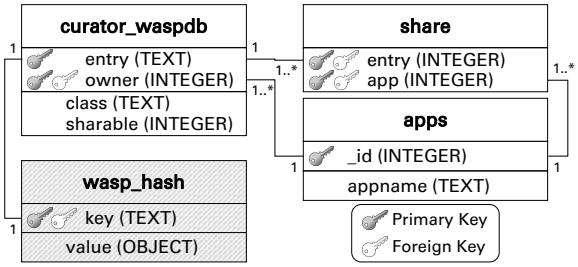


Figure 3: Schema for the WaspDB-based CURATOR.

permissions, the data is queried, decrypted, and forwarded to the caller's wrapper which handles the deserialization. Then, the object is available for the caller.

For the access control and the automatic deserialization, CURATOR requires additional maintenance data similar to the meta data of the SDC and MetaService.

4 IMPLEMENTATION

The schema applied in the SDC is shown in Figure 2. For each *key-value* entry, the *owner* and the release level (*shared*) are stored. In *share*, the tuple-based access rights are managed. This basic schema can be specialized towards a certain data type by adding further *value* columns—one column per attribute of the data type. Since SDC's schema does not provide meta data required for automatic deserialization, we extend it by MetaService's type casting maintenance data.

4.1 Technology Discussion

Since SDC's relational data schema is not flexible enough to handle heterogenous data types—either the SDC has to store several attributes in a single *value* column or a specialized schema is required for every data type—we consider three alternatives for CURATOR. Relational databases have performance issues when handling big data objects [38]. So, the usage of NoSQL databases seems to be a better strategy. We assess the two best NoSQL database types in terms of flexibility and performance, namely key-value stores (1) and document stores (2) [27]. For comparison, we also introduce also a relational approach storing data as BLOBs (3). The performance of the three approaches is evaluated in Section 5.

(1) Key-Value Store. The most obvious approach is the usage of a key-value store. Unlike the relational key-value schema of the SDC, a key-value store supports values of any data type, whereas the SDC has to declare a fixed type for the value column and any stored data has to be decomposed accordingly.

WaspDB [25] is one of the fastest and most resource-efficient key-value stores for Android. It accepts any kind of Java object as both, key and value. The serialization is handled by the *Kryo* framework[4]. WaspDB creates on-disk hashmaps, the so-called *Wasp Hashes*, in which the key-value pairs are stored. Additionally, WaspDB supports AES256 encryption.

Despite the high similarity to the SDC's relational approach, crucial adaptations are required concerning the data schema. WaspDB is incapable to store additional meta data for each key-value pair. However, this is necessary for CURATOR, on the one hand for the access rights management and on the other hand for registering the data types, which is required by the CURATOR Wrapper Library for deserialization. Figure 3 shows the adapted schema. Thus, this meta data has to be stored separately, e. g., in a SQLite database. In *curator_waspdb*, the WaspDB's key is used as a foreign key to link the two databases. The maintenance data is managed analogous to the one of the SDC.

(2) Document Store. As CURATOR operates on Java objects, also the storage as JSON documents seems reasonable. In a benchmark test, the document store *Couchbase* outperforms *Cassandra* and *MongoDB* in terms of latency and throughput [39]. There is a mobile version called *Couchbase Lite* [23]. It works either standalone (i. e., data is stored on the smart device) or it syncs with a remote backend. Data is passed to Couchbase as a hashmap, i. e., objects have to be decomposed into its attributes apriori. Indexes can be created to enable querying for certain attributes. Data is encrypted using AES256.

Listing 1 shows CURATOR's data schema for Couchbase. Maintenance data is stored directly in the document. It comprises the same information as the WaspDB-based schema. This data is attached (or removed respectively) by the Wrapper Library during (de)serialization. The actual payload is given as a key-value pair for each of the object's attributes.

(3) Relational Database. As studies show that NoSQL databases for smart devices perform poorly compared to Android's SQLite database in some cases [24], we also consider a SQLite-based implementation. Figure 4 shows its data schema. It is quite similar to the SDC's schema with two key differences. Objects are stored as serialized *BLOB*s (*payload*) and the maintenance data is extended

```
CuratorDocument {
  //Maintenance Data for CURATOR
  _id (long): Internal Identifier,
  _class (String): Data Type,
  _owner (String): Document Owner,
  _sharable (boolean): is Sharable?,
  _share (List): Apps with Access Rights,
  //Actual Payload
  attribute_1 (any): First Attribute,
  ...
}
```

Listing 1: Schema for the Couchbase-based CURATOR.

[4] https://github.com/EsotericSoftware/kryo/

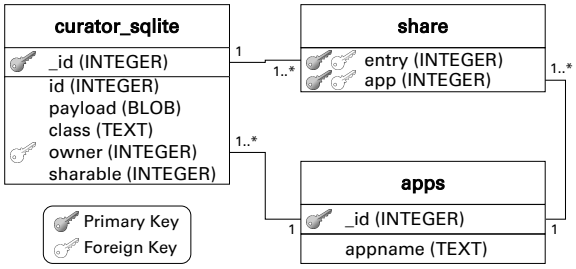


Figure 4: Schema for the SQLite-based CURATOR.

by a *class* column for object's data type. Based on this, an object can be processed directly without having to decompose it into its attributes. As SQLite supports no encryption by default, we store each object as *SealedObject*[5].

4.2 Technical Realization

The SDC is implemented as an extension of the *Privacy Management Platform (PMP)* [35, 34]. The PMP is a privacy-aware data provisioning system. Any kind of data source can be included as so-called *Resources*. The PMP enables apps to query these Resources. This includes also authentication and fine-grained authorization features. PMP Resources are implemented as Android apps, i. e., the Android platform does not have to be manipulated in order to add further Resources. For details on the PMP, please refer to literature (see [35, 34]). Because of the PMP's data provisioning and security features proved beneficial for the SDC, we decided to implement CURATOR as a PMP Resource likewise.

Figure 5 outlines how CURATOR is implemented. Apps need to include the *PMP Library* in order to use the PMP. Among others, this library contains the registration function which is required for authorization towards the PMP. We included the Wrapper Library into this library (*a*). When a new Resource is added, the PMP notifies all apps about the available Resources and their *APIs* (*b*). An app puts its queries directly to the PMP and the PMP verifies whether the app has the required permissions (*c*). Only valid requests are forwarded to CURATOR. For these, a second authorization check is executed within the CURATOR Resource. The *Access Management* verifies for each query, whether the inquiring app is allowed to access the requested objects—i. e., is the app the data owner or is the data shared explicitly with this app (*d*). Legit queries are passed to *Storage Abstraction* which translates the query into an expression that is evaluable by the data stores (*e*). There, the query is executed and the PMP forwards the results to the inquiring app (*f*).

[5]see https://goo.gl/FPPQev

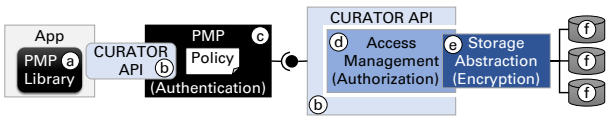


Figure 5: PMP-based Implementation of CURATOR.

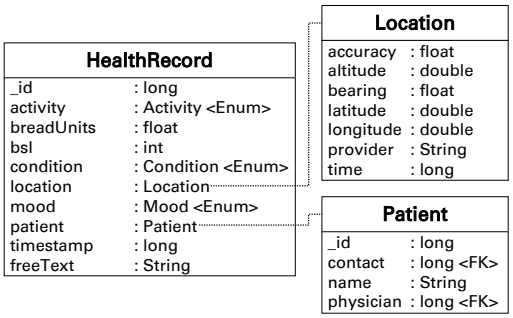


Figure 6: eHealth Data Model Used for Evaluation.

Moreover, the PMP monitors Android's uninstall process [31]. That way, the PMP informs CURATOR whenever an app is uninstalled, whereupon CURATOR deletes all data owned by this app. So, the stored data cannot outlive its owner.

5 EVALUATION

For the performance evaluation, we chose a use case from the health domain. *Secure Candy Castle* [37, 32] is a mobile health game for children suffering from diabetes. The game motivates young patients to carry out the required measurements regularly. Each reading is augmented by the location the measurement took place. These data are stored in conjunction with details about the patient's condition and his or her eating behavior in an electronic diabetes diary. Figure 6 shows the applied data model for a single diary entry. The actual health data are stored in the *HealthRecord* class. For *activity*, *condition*, and *mood* a selection of predefined values are given in separated enumeration classes. The measurement location and patient details are stored as nested classes. If these entries are also available for other apps, users benefit from positive side effects. Since the surrounding has an impact on a patient's condition (e. g., air pollution or noise-based stress), a navigation app could consider the user's current condition in order to find the healthiest way [16]. Thus, the management of HealthRecord objects is a realistic use case.

We compare the performance of CURATOR with the performance of Android's Content Provider approach (with a SQLite database) and the SDC (with AES256). On behalf of CURATOR, we differentiate between its WaspDB-based, Couchbase-based, and SQLite-based implementation. For the Content Provider's database and the SDC, we mapped each HealthRecord attribute (and its nested objects) to separate database columns. Our benchmark suite consists of two separated benchmarks which are executed mutually: The *writing benchmark* creates n random HealthRecord objects (with $n \in \{500, 1000, 2000, 4000, 8000, 16000, 32000, 64000\}$). Each entry is allocated with a different location and patient. Then, the objects are stored in the respectively evaluated data store and shared with the *reading benchmark* app[6]. Following this, the reading benchmark requests the n objects in random order. After each writing-reading cycle, the database and its cache is cleared in order to prevent any influences by warm caches. This cycle is repeated

[6]By default, Content Providers share data with any app.

Table 1: Evaluation Results Overview

	Writing Benchmark					*Reading Benchmark*				
Datasets	**AND**	**SDC**	**CUR$_W$**	**CUR$_C$**	**CUR$_S$**	**AND**	**SDC**	**CUR$_W$**	**CUR$_C$**	**CUR$_S$**
Overall Runtime in Seconds										
500 Tuples	191.1	45.8	46.3	38.3	31.9	85.1	37.3	40.7	34.4	31.1
64,000 Tuples	5,239.1	5,397.9	13,449.8	4,775.5	4,177.5	11,236.2	15,196.4	12,294.8	4,367.4	22,057.9
Total Battery Drain in Milliampere per Hour										
500 Tuples	12.65	3.28	3.45	2.52	2.40	5.62	2.78	2.92	2.23	2.19
64,000 Tuples	339.97	378.05	939.13	308.82	292.48	770.55	1,123.47	851.61	292.94	1,967.18
Average CPU Usage in Percentage Terms										
500 Tuples	15.2	12.8	25.6	21.0	15.2	26.5	26.0	25.7	24.8	26.7
64,000 Tuples	17.6	14.5	24.8	20.0	15.4	26.0	25.4	25.6	24.9	25.2
Maximum Memory Usage in Megabyte (PSS)										
500 Tuples	28.58	24.26	29.29	29.39	24.02	20.74	26.69	29.09	31.78	26.39
64,000 Tuples	72.54	68.47	69.85	70.60	72.18	54.35	67.09	71.47	71.67	67.35
Maximum Memory Usage in Megabyte (Private Dirty)										
500 Tuples	25.62	20.97	25.89	25.23	21.04	18.02	23.34	25.59	27.52	23.26
64,000 Tuples	69.00	64.85	66.03	66.87	68.21	51.39	64.42	67.79	67.76	64.34
Total Database Size in Kilobyte										
500 Tuples	217	284	193	846	1,020	217	284	193	846	1,020
64,000 Tuples	21,496	28,148	21,507	86,428	128,488	21,496	28,148	21,507	86,428	128,488

AND – *Content Provider Approach*; **SDC** – *Secure Data Container*; **CUR$_W$** – *WaspDB-based CURATOR*; **CUR$_C$** – *Couchbase-based CURATOR*; **CUR$_S$** – *SQLite-based CURATOR.*

for 10 times and the median is calculated. Then, n is increased progressively. We record overall runtime, total battery drain, average CPU workload, maximum memory usage[7], and database sizes.

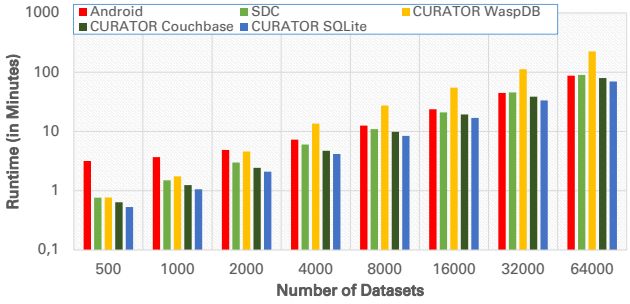
The evaluation is carried out on a Motorola Moto E2 Phone (Qualcomm Snapdragon 410 CPU; 1 GB RAM; 2,390 mAh battery capacity) which is a lower class smartphone. On the phone runs a Vanilla-Android 5.1.1 (kernel version 3.10.87).

The evaluation results for the smallest and biggest dataset are shown in Table 1. In the following, we discuss the key findings: **[A]** Concerning the **overall runtime**, the creation and usage of Content Provider causes a fixed overhead. Therefore, this approach is substantially slower for small dataset. From approximately 3,000 datasets on, this overhead amortized by the rising encryption costs of the other approaches in the writing benchmark. In the reading benchmark, the amortization point is reached at approximately 32,000 datasets. Only the Couchbase-based CURATOR is always significantly faster. All in all, the encryption costs can be virtually neglected. **[B]** The findings concerning the **total battery drain** are very similar to the ones observed for the overall runtime. From a user's point of view, the runtime and the battery drain are the most important qualities. The bad results of the SQLite-based CURATOR for big n in the reading benchmark are explainable since the mobile version of SQLite is not fitted for working with big databases [38].

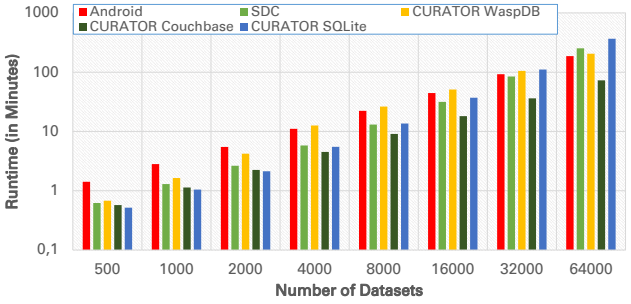
[7]We consider proportional set size and private dirty pages.

[C] The **average CPU usage** for the writing benchmark is consistently low for the relational-based approaches, whereas the two NoSQL approaches have higher costs. By contrast in the reading benchmark, all approaches are alike. **[D]** The **maximum memory usage** considers memory peaks during the benchmarks, only. The proportional set size (PSS) comprises all of a process's RAM pages. Private dirty calculates the fraction that is not shared with other processes. Our benchmark shows that the approaches behave similar in both aspects. Therefore, we do not differentiate between those two metrics in the following. In the writing benchmark, the relational-based approaches consume less memory. The usage of Content Provider causes an additional overhead in the writing benchmark, while it requires the least memory in the reading benchmark. Apart from that, the relational-based approaches beat the NoSQL approaches in terms of memory consumption. **[E]** Concerning the **database size**, the SDC allocates more storage space compared to the Content Provider approach due to its additional maintenance data. The usage of BLOBs in the SQLite-based CURATOR leads to the biggest database size (closely followed by Couchbase), whereas WaspDB has the most compact storage technique.

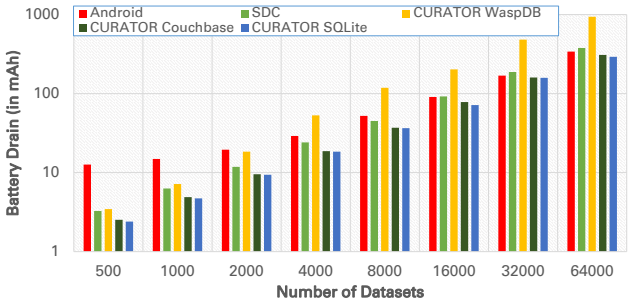
The key figures of our evaluation for exponentially increasing datasets concerning the overall runtime and the total battery drain are depicted in Figure 7. Keep in mind, that the Content Provider approach does not secure its data, whereas the other four approaches apply AES256-based data encryption. I. e., the usage of a Couchbase-based CURATOR is not only more performant, but also more secure.



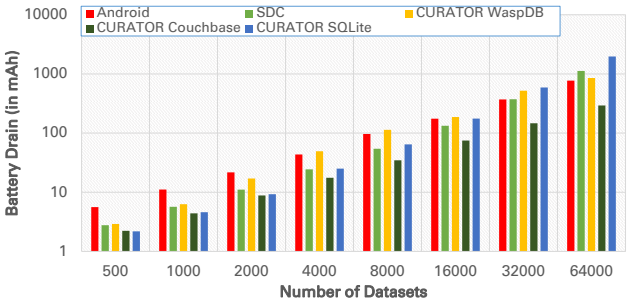
(a) Overall Runtime (Writing Benchmark)



(b) Overall Runtime (Reading Benchmark)



(c) Total Battery Drain (Writing Benchmark)



(d) Total Battery Drain (Reading Benchmark)

Figure 7: Details on the Most Relevant Evaluation Results.

Table 2 summarizes in detail for Android's Content Provider approach, the SDC, and CURATOR to what extent they meet the seven essential characteristics regarding manageability, security and performance. This involves (1) how easy the respective approach can be used by an app, (2) whether heterogeneous data types are supported, (3) whether data security is taken into account, (4) whether access to the data can be restricted, and (5) how resource-saving the respective approaches are. This shows that only CURATOR fully meets all these requirements. The insignificantly higher memory consumption is a minor issue for mobile platforms.

Table 2: Feature Comparison Table

Feature	AND	SDC	CUR
Manageability	◔	◕	●
Heterogeneous Data Support	◑	◑	●
Data Security	○	●	●
Access Control	◔	●	●
Battery Efficiency	○	◑	●
Memory Efficiency	◑	●	◕
Storage Efficiency	●	◕	●

AND – *Content Provider Approach*; **SDC** – *Secure Data Container*; **CUR** – *CURATOR.*

Lessons Learned.

§ 1 CURATOR's security features cause no performance losses.

§ 2 The SQLite-based CURATOR is recommendable for writing operations whereas the Couchbase-based CURATOR is best for reading operations in terms of its runtime and battery drain.

§ 3 NoSQL-based CURATOR implementations have a higher CPU usage.

§ 4 NoSQL-based CURATOR implementations require more main memory.

§ 5 WaspDB has the most compact storage technique.

§ 6 Due to storage abstraction, the best database technology can be applied according to given hardware constraints. Thereby, CURATOR is superior in any category to both, Content Provides and the SDC.

6 CONCLUSION AND OUTLOOK

Today, smart devices are popular as never before. Due to manifold apps, they are useful in any situation. To unlock their full potential, a manageable, secure, and performant data exchange mechanism is required. As no mobile platform implements such a mechanism, we address this issue.

We come up with a concept of a **se**c**ur**e **s**h**a**red obje**c**t s**t**o**r**e called *CURATOR*. Any app can pass their data as Java objects to CURATOR to store them as well as share them with other apps. In

CURATOR various security features are applied including authentication, authorization, and encryption. Storage abstraction enables to substitute the data store applied in CURATOR. We consider key-value stores, document stores, and relational databases. We implement CURATOR with each of the three technologies and evaluate its respective performance. The evaluation results show that CURATOR outperforms both, Android's insecure data exchange mechanism and the SDC [36].

In order to exchange data between smart devices, a distributed storage infrastructure similar to Mobius [5] is required. The *PATRON* research project[8] investigates, how CURATOR can be applied in such a setting.

ACKNOWLEDGMENTS

This paper is part of the PATRON research project which is commissioned by the Baden-Württemberg Stiftung gGmbH. The authors would like to thank the BW-Stiftung for the funding of this research.

REFERENCES

[1] Rakesh Agrawal et al. 2008. The Claremont Report on Database Research. *ACM SIGMOD Record*, 37, 3, 9–19.
[2] Melissa Chau et al. 2017. Smartphone OS Market Share, 2017 Q1. International Data Corporation (IDC). http://www.idc.com/promo/smartphone-market-share/os. (May 2017).
[3] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (MobiSys '11), 239–252.
[4] Hwayoung Choe, Jihun Baek, Hoeheon Jeong, and Sangwon Park. 2011. MetaService: An Object Transfer Platform Between Android Applications. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (RACS '11), 56–60.
[5] Byung-Gon Chun, Carlo Curino, Russell Sears, Alexander Shraer, Samuel Madden, and Raghu Ramakrishnan. 2012. Mobius: Unified Messaging and Data Serving for Mobile Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (MobiSys '12), 141–154.
[6] Nikolay Elenkov and Yaniv Rodenski. 2014. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, San Francisco, CA, USA.
[7] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. Understanding Android Security. *IEEE Security and Privacy*, 7, 1, 50–57.
[8] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (SOUPS '12), 3:1–3:14.
[9] Google Inc. 2017. Android Things. Android Things. https://developer.android.com/things. (July 2017).
[10] Google Inc. 2016. Content Providers. Android Developers. https://developer.android.com/guide/topics/providers/content-providers.html. (Dec. 2016).
[11] Google Inc. 2017. Copy and Paste. Android Developers. https://developer.android.com/guide/topics/text/copy-paste.html. (May 2017).
[12] Google Inc. 2017. Intent. Android Developers. https://developer.android.com/reference/android/content/Intent.html. (July 2017).
[13] Google Inc. 2016. Storage Options. Android Developers. https://developer.android.com/guide/topics/data/data-storage.html. (Nov. 2016).
[14] Behnaz Hassanshahi and Roland H.C. Yap. 2017. Android Database Attacks Revisited. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (ASIA CCS '17), 625–639.
[15] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting Storage for Smartphones. *ACM Transactions on Storage (TOS)*, 8, 4, 14:1–14:25.
[16] Martin Knöll, Yang Li, Katrin Neuheuser, and Annette Rudolph-Cleff. 2015. Using space syntax to analyse stress ratings of open public spaces. In *Proceedings of the 10th International Space Syntax Symposium* (SSS '15), 123:1–123:15.
[17] Shin-ya Nishizaki, Masayuki Numao, Jaime D. L. Caro, and Merlin Teodosia C. Suarez, (Eds.) 2017. *Securing Health Information System with CryptDB. Theory and Practice of Computation: Proceedings of Workshop on Computation: Theory and Practice WCTP2015*. World Scientific, Singapore, Hackensack, NJ, London, 136–158.

[8] see http://patronresearch.de

[18] Fabio Martinelli, Paolo Mori, and Andrea Saracino. 2016. Enhancing Android Permission Through Usage Control: A BYOD Use-case. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (SAC '16), 2049–2056.
[19] Charlie Miller. 2011. Mobile Attacks and Defense. *IEEE Security and Privacy*, 9, 4, 68–70.
[20] Mark L. Murphy. 2017. *The Busy Coder's Guide to Android Development*. CommonsWare, LLC, Pennsylvania, USA.
[21] Simone Mutti, Enrico Bacis, and Stefano Paraboschi. 2015. SeSQLite: Security Enhanced SQLite: Mandatory Access Control for Android Databases. In *Proceedings of the 31st Annual Computer Security Applications Conference* (ACSAC '15), 411–420.
[22] Machigar Ongtang, Kevin Butler, and Patrick McDaniel. 2010. Porscha: Policy Oriented Secure Content Handling in Android. In *Proceedings of the 26th Annual Computer Security Applications Conference* (ACSAC '10), 221–230.
[23] 2015. *Couchbase Lite on Android. Pro Couchbase Server*. Apress, Berkeley, CA. Chap. 14, 307–317.
[24] Trevor Perrier and Fahad Pervaiz. 2013. NoSQL in a Mobile World: Benchmarking Embedded Mobile Databases. Tech. rep. University of Washington.
[25] rehacktive. 2016. WaspDB. GitHub. https://github.com/rehacktive/waspdb. (Nov. 2016).
[26] Daniel Schreckling, Joachim Posegga, Johannes Köstler, and Matthias Schaff. 2012. Kynoid: Real-time Enforcement of Fine-grained, User-defined, and Data-centric Security Policies for Android. In *Proceedings of the 6th IFIP WG 11.2 International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems* (WISTP '12), 208–223.
[27] Ben Scofield. 2010. NoSQL: Death to Relational Databases(?) Talk at CodeMash.
[28] Hossain Shahriar and Hisham M. Haddad. 2014. Content Provider Leakage Vulnerability Detection in Android Applications. In *Proceedings of the 7th International Conference on Security of Information and Networks* (SIN '14), 359:359–359:366.
[29] Faisal Shahzad, Waheed Iqbal, and Fawaz S. Bokhari. 2015. On the use of CryptDB for securing Electronic Health data in the cloud: A performance study. In *Proceedings of the 2015 17th International Conference on E-health Networking, Application Services* (HealthCom '15), 120–125.
[30] Dan Siewiorek. 2012. Generation Smartphone. *IEEE Spectrum*, 49, 9, 54–58.
[31] Christoph Stach. 2015. How to Deal with Third Party Apps in a Privacy System — The PMP Gatekeeper. In *Proceedings of the 2015 IEEE 16th International Conference on Mobile Data Management* (MDM '15), 167–172.
[32] Christoph Stach. 2016. Secure Candy Castle — A Prototype for Privacy-Aware mHealth Apps. In *Proceedings of the 2016 IEEE 17th International Conference on Mobile Data Management* (MDM '16), 361–364.
[33] Christoph Stach and Bernhard Mitschang. 2015. Der Secure Data Container (SDC) – Sicheres Datenmanagement für mobile Anwendungen. *Datenbank-Spektrum*, 15, 2, 109–118. In German.
[34] Christoph Stach and Bernhard Mitschang. 2014. Design and Implementation of the Privacy Management Platform. In *Proceedings of the 2014 IEEE 15th International Conference on Mobile Data Management* (MDM '14), 69–72.
[35] Christoph Stach and Bernhard Mitschang. 2013. Privacy Management for Mobile Platforms – A Review of Concepts and Approaches. In *Proceedings of the 2013 IEEE 14th International Conference on Mobile Data Management* (MDM '13), 305–313.
[36] Christoph Stach and Bernhard Mitschang. 2016. The Secure Data Container: An Approach to Harmonize Data Sharing with Information Security. In *Proceedings of the 2016 IEEE 17th International Conference on Mobile Data Management* (MDM '16), 292–297.
[37] Christoph Stach and Luiz Fernando M. Schlindwein. 2012. Candy Castle — A Prototype for Pervasive Health Games. In *Proceedings of the 2012 IEEE International Conference on Pervasive Computing and Communications Workshops* (PerCom '12), 501–503.
[38] Dam Quang Tuan, Seungyong Cheon, and Youjip Won. 2016. On the IO Characteristics of the SQLite Transactions. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* (MOBILESoft '16), 214–224.
[39] Frank Weigel. 2012. Benchmarking Couchbase. Talk at CouchConf.
[40] Sven Dietrich, (Ed.) 2014. *Attacks on Android Clipboard. Detection of Intrusions and Malware, and Vulnerability Assessment: 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings*. Springer International Publishing, Cham, 72–91.
[41] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. 2016. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *Proceedings of the Network and Distributed System Security Symposium 2016* (NDSS '16), 1–15.
[42] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium* (NDSS '13), 1–16.