# Querying Invisible Objects: Supporting Data-Driven, Privacy-Preserving Distributed Applications

Yin Liu, Zheng Song, and Eli Tilevich
Software Innovations Lab
Virginia Tech
{yinliu,songz,tilevich}@cs.vt.edu

## ABSTRACT

When transferring sensitive data to a non-trusted party, end-users require that the data be kept private. Mobile and IoT application developers want to leverage the sensitive data to provide better user experience and intelligent services. Unfortunately, existing programming abstractions make it impossible to reconcile these two seemingly conflicting objectives. In this paper, we present a novel programming mechanism for distributed managed execution environments that hides sensitive user data, while enabling developers to build powerful and intelligent applications, driven by the properties of the sensitive data. Specifically, the sensitive data is never revealed to clients, being protected by the runtime system. Our abstractions provide declarative and configurable data query interfaces, enforced by a lightweight distributed runtime system. Developers define when and how clients can query the sensitive data's properties (i.e., how long the data remains accessible, how many times its properties can be queried, which data query methods apply, etc.). Based on our evaluation, we argue that integrating our novel mechanism with the Java Virtual Machine (JVM) can address some of the most pertinent privacy problems of IoT and mobile applications.

## KEYWORDS

Data Privacy, Data-Intensive Applications, Programming Abstractions, Virtual Machine Design

**ACM Reference format:**
Yin Liu, Zheng Song, and Eli Tilevich. 2017. Querying Invisible Objects: Supporting Data-Driven, Privacy-Preserving Distributed Applications. In *Proceedings of MANLANG'17, Czech Republic, Sep. 25–29, 2017, Prague,* 13 pages.
DOI: 10.475/123_4

## 1 INTRODUCTION

Mobile, IoT, and wearable devices continuously collect increasing volumes of user data, much of it sensitive. Health monitors track their owners' vital signs; smart phones read sensory personal data, including GPS location, velocity, direction, etc.; IoT devices obtain their environmental information. When it comes to sensitive data, there is a fundamental conflict between end-user requirements and application developer aspirations. End-users want to make sure that their sensitive data remains private, inaccessible to non-trusted parties. Application developers want to leverage the sensitive data's properties to provide intelligent applications that provide personalized user experiences and intelligent, context-sensitive services. These two objectives are seemingly irreconcilable.

Consider the following three examples of data-intensive applications that handle potentially sensitive data.

(I.) Mobile navigation applications provide real-time traffic information to their users, who also contribute this information when using the applications. When providing navigation services, a navigation application continuously uploads to the cloud its device's current GPS location and speed, which are then used to estimate real-time traffic information. Although this information is uploaded anonymously, given enough GPS data and speed, a non-trusted party may be able to learn about the device owner's daily routine and abuse this information for nefarious purposes. Therefore, although the contributors may be willing to help estimate real-time traffic information, they would not want their GPS location revealed and recorded.

(II.) A group of friends is looking for a restaurant to dine together. Their smartphones maintain their owners' dining histories. Based on the dining history of each individual, the edge server at a shopping plaza can suggest which restaurants would be most suitable for the entire group. However, the individuals may be unwilling to share their raw dining histories with a non-trusted party.

(III.) A smart building may adjust its temperatures and lighting levels, as driven by the preferences of its current occupants. Users wearing personal health trackers and equipped with smartphones can report their owners' vitals and building location. This information is one of the key pieces that make a building "smart." However, individuals may not want to reveal their vitals and current location to a non-trusted party.

Please, notice that in all three examples above, intelligent services can still be provided while keeping the users' sensitive data invisible. To calculate the average reported velocity, the traffic monitoring system should not need to be aware of the actual velocity of each passing vehicle. This sensitive information can remain hidden, and only its statistical average calculated and used in predicting current traffic conditions. To recommend a restaurant, the edge server should not need to know which specific restaurants the individuals involved have patronized in the past. Instead, it can suggest a mutually acceptable restaurant, based on each individual's most favored cuisine, statistical information that can be obtained by querying the dining histories. To intelligently adjust its environmental settings, a smart building can remain unaware


*MANLANG'17, Czech Republic*

DOI: 10.475/123_4

of the building inhabitants' actual vitals or exact locations. It only needs the vital signs' statistical averages, as reported by each of its autonomous climate-controlled areas.

We posit that when developing data-intensive, privacy-preserving applications, developers often need to keep the sensitive data invisible, while being able to query that data for various properties. Both data privacy and the ability to query the data should be guaranteed by the runtime system. However, existing programming mechanisms provide no explicit support for this type of privacy preservation. In Object-Oriented programming, `private` fields can only be accessed by the `public` methods in the same class. These methods, however, are free to implement any access policy to sensitive data and even return the values of privately declared fields to the client. In addition, this method-level protection can be bypassed via Reflection [7]. If a programming mechanism can guarantee that the sensitive data will remain invisible, while providing a controlled way to query the data, this mechanism can support the development of emerging applications in the mobile and IoT domains, thereby improving their data privacy.

In this paper, we present Object Expiration (ObEx), a novel programming mechanism, supported by a lightweight, portable runtime system, that possesses all the privacy preservation properties described above. The key idea behind ObEx objects is that they securely store some sensitive data, to which they never provide access to their clients. In other words, the sensitive data remains hidden in the system layer and never revealed to the application that uses these objects. Although the sensitive data remains invisible, clients can execute various pre-defined queries (e.g., testing for equality, comparisons, membership in a range, etc.) against it. However, the total number of queries and the time limit over which the queries can be executed are limited by a declarative policy specified when instantiating ObEx objects. The runtime system ensures that the specified policy is preserved when ObEx objects are serialized and transferred across the network. When deep-copying one ObEx object to another by using serialization, the runtime keeps only one version of the sensitive data, causing the copies to become aliases to the same data. When transferring an ObEx object across the network, the runtime system encrypts its sensitive data, and starts the expiration timer as soon as the object is unserialized at the destination site.

This paper makes the following contributions:

(1) We introduce ObEx objects, a programming mechanism that supports the development of data-driven, privacy-preserving distributed applications, common in the emerging mobile, wearable, and IoT domains.
(2) We empirically evaluate ObEx objects in terms of their usability and efficiency with benchmarks and privacy threats.
(3) We present developer guidelines for using ObEx objects in a given application scenario, as informed by our empirical observations.

The remainder of this paper is structured as follows. Section 2 further explains how ObEx can be used. Section 3 provides the technical background for this research. Section 4 details the ObEx object programming model and the runtime system enabling it. After discussing how developers can use ObEx in their applications in Section 5, we describe how we evaluated the effectiveness and efficiency of ObEx in Section 6. We then discuss related work in Section 7. Section 8 presents conclusions and future work directions.

## 2 USAGE SCENARIO AND USE CASE

In this section, we first present a typical ObEx usage scenario. Then, we show how using ObEx can effectively reconcile conflicting requirements of data producers and consumers in a realistic use case.

### 2.1 Typical Usage Scenario

Consider a scenario of collecting and making use of some sensitive data. The data is sensitive in the sense that it contains some private information that should not be revealed to outside parties. The role of *a data producer* is to collect or generate some sensitive data. The role of *a data consumer* is to make use of the sensitive data's properties (without having the ability to access the data) to provide some intelligent services. As owners of sensitive data, producers also determine which access policy should be applied to the data, so as to preserve user privacy while permitting consumers to leverage some properties of the sensitive data.

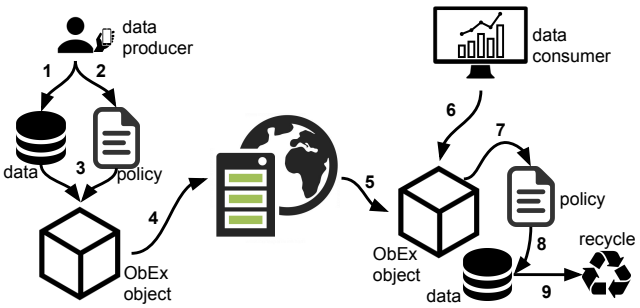


**Figure 1: Typical Usage Scenario.**

Data producers and data consumers operate in an environment of *mutual distrust*. Producers would not share sensitive data with consumers, both to preserve user privacy and to comply with privacy regulations. Consumers would not trust the results of any operations performed by producers over the data. Consumers need to be able to compute over the sensitive data on their own, as their computational procedures constitute their intellectual property (IP), which is not to be revealed to the producers.

Figure 1 shows a process of developing a distributed data-intensive application that preserves data privacy by means of ObEx. The process starts with the introduction of sensitive data into the system (step 1). The data is then stored in an ObEx object. When instantiating the object, the access policy determined by the producer (step 2) is passed as a parameter to the constructor (step 3). Then, the ObEx runtime system serializes, encrypts, and transfers the ObEx object to the data consumer, which is a non-trusted party that should be unable to access the sensitive data (step 4). Serializing data into a binary string, Step 4 can be easily embedded into the data marshaling/unmarshaling processes of standard web services. When the ObEx object arrives to the non-trusted party's site, the ObEx runtime decrypts and unserializes the transferred object, making

it ready for client queries (step 5). The access policy dictates the type of queries, their number, and the total time over which they can be made. The runtime enforces the policy by keeping track of the queries made and the time elapsed (step 7). Once either of the thresholds is reached, the runtime reliably clears the sensitive data associated with the ObEx object (step 8 and 9). Once the data is cleared, all subsequent queries result in a runtime exception, raised by the ObEx runtime.

## 2.2 Use Case

A recent sociological study analyzed the integration and behavioral patterns of people who have recently moved to live in the city of Shanghai [24]. For the study, China Telecom supplied anonymized metadata from 698 million of traces of all its Shanghai customers. In this demographic study, researchers analyzed the provided call traces and concluded that city "locals" and "migrants" behaved dissimilarly with respect to their daily itineraries and calling patterns. It was surprising how many behavioral patterns the researchers were able to infer, given that the provided metadata only included the age, sex, and call traces of the phone customers. For example, while the locals tend to communicate with people of their own age, the migrants tend to communicate with people of various ages. As a result, by analyzing the age of one's contacts in the trace, one can determine whether the subject is likely to be a migrant.

The study also raised some questions from civil libertarians, which worry that by disclosing even anonymized data about phone customers, phone companies enable third parties to learn sensitive information about people's lives, thus violating their privacy. Please, notice, however, that the researchers, which were given access to the anonymized metadata, can be considered *a trusted party*, someone who would not exploit the sensitive data for nefarious purposes. For these demographic researchers, the sensitive data is only a tool that enables them to infer the integration patterns of migrants moving to a new city.

However, commercial entities could leverage the age and sex information to create useful and intelligent applications and services, but additional care must be taken to preserve user privacy. Companies developing IoT and mobile applications can target certain demographics, in which "possible migrants" would be a large group with distinct interests and needs. For example, the mobile ads delivered to this group can prominently feature 'apartments for rent' information. Similarly, newcomers would appreciate more detailed guidance from navigation applications.

How can one enable software developers to leverage the sensitive data of the phone company's customers, while ensuring that the customers' privacy is fully preserved? In other words, we want developers to be able to infer which customers are *likely migrants*, without being able to infer their other behavioral patterns. Next, we show how using ObEx can reconcile the objectives of creating intelligent mobile services and preserving user privacy.

As in the typical scenario described above, the telecoms operator is the data producer, while the third-party services for the mobile devices of phone customers are data consumers. These services can provide customized business intelligence to the consumers' mobile/IoT applications. To realize this requirement, the data consumers should be able to analyze the age of the contacts of a smartphone customer, while being unable to access their actual age values, which can be used to infer sensitive information, such as the smartphone customer's own age. Instead, they should be able to perform statistical analysis on a collection of ages, calculating its count, average, median, mode, and standard deviation. Assume that age is represented as an integer. Now, consider how one can put the aforementioned policy in place by using ObEx. Next, we describe the three main steps of the process: (1) data collection, (2) data transfer, and (3) data analysis.

**(1) Data Collection** The data collection step wraps sensitive data in ObEx objects and instructs the ObEx runtime system to transfer the objects to the consumers. Recall that when creating an ObEx object, an access policy must be given. In this case, the producer can define an access policy with the following parameters (two queries of any kind, 10,000 milliseconds lifetime, all query methods permitted with the exception for `sum`). The access policy has to be consistent with the needs of data consumers; otherwise, the provided ObEx objects are of no use to the consumers. Section 4 provides details about the ObEx APIs and runtime design.

**(2) Data Transfer** The data transfer process is responsible for moving initialized ObEx objects across the network to the consumer sites. To that end, the ObEx runtime encrypts and serializes the sensitive data to a binary stream. The stream is transferred to the destination site, at which the runtime unserializes, decrypts, and reconstructs the stream into an ObEx object instance. The lifetime timer starts immediately after an ObEx object is reconstructed. Section 4 describes the ObEx distributed runtime.

**(3) Data Analysis** The data analysis process is concerned with executing queries against the sensitive data protected by ObEx objects. In this use case, the telecoms provides a collection of ObEx objects containing the ages of a given customer's contacts. Recall that the access policy allows data consumers to invoke any statistical method with the exception of `sum`, with only two queries permitted. A mobile app can invoke the `average` and `stdDeviation` methods to obtain the necessary information required to be able to make a reasonable guess whether the customer is likely a migrant. Notice, that ObEx provides the necessary information without revealing the actual ages of the customer's contacts. Furthermore, the sensitive information becomes inaccessible after 10 seconds, thus further preventing potential abuse by nefarious parties. Section 4 details the ObEx APIs, while Section 6 discusses the performance characteristics of our reference implementation.

To sum up, the raison d'être behind ObEx is to resolve the inherent conflict between the user's privacy needs and the mobile developer's wishes when it comes to managing sensitive data. In this use case, the ObEx protection ensures that the actual ages of phone customers are never revealed to non-trusted parties, which in this case are services supporting mobile applications. These services can still obtain valuable business intelligence by querying ObEx objects, without directly accessing the sensitive data. The following sections discuss the technical details of the ObEx design, implementation, and evaluation.

# 3 BACKGROUND

In this section, we briefly present several known risks to data privacy, which motivate our work. Then, we outline the most commonly known vulnerabilities of Java objects.

## 3.1 Risks to Data Privacy

Retaining and transferring the user's sensitive data without a suitable protection mechanism gives rise to a large number of serious privacy vulnerabilities. Among the top ten mobile security risks, as per the 2016 report by the Open Web Application Security Project (OWASP) [1], are data vulnerability, insecure communication and unreliable authentication procedures. The Common Vulnerabilities and Exposures (CVE) reports a lot of known relevant exploits. For example, due to the man-in-the-middle attack, the Eview EV-07S GPS Tracker exposes lots of sensitive data, such as the current GPS location and IMEI numbers, when transmitting data over the Internet [2]. In the meantime, users' location is disclosed on the website because the Sleipnir Mobile application misapplies Geolocation API and sends the sensitive data without gaining user permission [3]. What is worse is that, whether shared legitimately or inappropriately, sensitive data can be stored persistently. Attackers can then use that sensitive data to perpetrate a variety of subsequent privacy exploits.

## 3.2 Exploits of Java Objects

Several known exploits can compromise Java objects containing sensitive data. Malicious accesses to Java objects have threatened a large number of applications [4] [5] [6]. The built-in Java language protection mechanisms, such as object encapsulation and garbage collection can be insufficient to defend against particularly elaborate attacks, particularly in distributed environments.

*3.2.1 Object Encapsulation.* One of the fundamental concepts of object-oriented programming (OOP) is encapsulation, which hides the sensitive data and behavior from object clients. Moreover, Java provides access modifiers to ensure data privacy. By applying the keyword `private` to a field, programmers expect the field not to be accessible from outside of its declaring class. The protection afforded by Java access modifiers can be bypassed by using reflection. With the right permission, an attacker can use reflection to directly access and modify `private` fields and invoke `private` methods. To help prevent this attack, the security manager mechanism [10] has been added to Java, but its effectiveness depends on all components being properly configured, a requirement that can be hard to fulfill in complex distributed systems. Proper configuration and deployment practices can prevent these attacks, but they require a universal adherence.

*3.2.2 Object Life Cycle.* When a programming object containing sensitive data goes out of scope, it becomes available for garbage collection. The actual collection time and scope are, however, entirely the prerogative of the collection policy in place. In some cases, however, waiting for the garbage collector to clear sensitive data may be insufficient. Instead, the sensitive data may need to be reliably cleared after reaching a certain threshold, as defined by the specified per-object access policy.

[1] Open Web Application Security Project Mobile Top 10 https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10
[2] CVE-2017-5239 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5239
[3] CVE-2014-0806 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0806
[4] CVE-2009-1084 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1084
[5] CVE-2009-2747 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2747
[6] CVE-2012-0393 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0393

# 4 DESIGN AND IMPLEMENTATION

In this section, we first introduce key design ideas behind ObEx objects, then we present the technical details of our implementation.

## 4.1 Design Overview

In the typical usage scenario described above, the inherent conflict between data producers and consumers is that the former require that their sensitive data be kept private, while the latter wish to leverage some properties of the private data to provide better services and build more intelligent applications. The ObEx mechanism reconciles these conflicting requirements. It keeps the sensitive data invisible, but enables non-trusted clients to query the data in a controlled way. Once the number of queries or the time limit is exceeded, the ObEx runtime clears the sensitive data, making it inaccessible for any future operations.

*4.1.1 Invisible Data.* The ObEx architecture keeps its sensitive data at the native layer, never passing it to the managed code (i.e., bytecode) layer. Hence, the sensitive data cannot be copied to regular Java objects. The ObEx API provides methods for serializing and cloning ObEx objects. However, these methods' implementation keeps the sensitive data secure at the native layer. Application clients can invoke any of the predefined ObEx query methods allowed by the policy in place. However, the methods are `native`, being executed in the native layer. In other words, although the data is invisible, not being accessible to non-trusted consumers, they still are able to obtain valuable insights about the data's properties (e.g., perform statistical calculations) that can enrich the functionality and user experience of applications and services.

*4.1.2 Configurable Expiration Policy.* How the sensitive data can be accessed is dictated by a configurable expiration policy that describes three interconnected limitations: max number of accesses, time-to-expiration, and available query methods. The first two limitations determine when the object and its sensitive data should be cleared. The last limitation confirms which query methods non-trusted clients are allowed to invoke.

In addition to query methods, the ObEx API includes meta-query methods, through which the developer can discover what type of data encapsulates a given ObEx object and which query methods can be invoked on the object. In essence, the meta-query methods return information about the access policy, whose restrictions have to be expressed by the producer.

*4.1.3 Enforceable Lifecycle.* Figure 2 shows the enforceable lifecycle of ObEx objects. In phase 1, although the sensitive data is hidden in the native layer, non-trusted clients can query the data from the managed layer. In phase 2, the object has cleared itself in the native layer as driven by a given policy, nullifying the data and disabling all future queries. In phase 3, the object has been completely garbage collected both in the runtime layer and the managed

layer. Currently, our approach supports the lifecycle enforcement in phases 1 and 2. However, in phase 3, the garbage collector only collects the managed portion of ObEx objects; the native layer's sensitive data is cleared based only on the access policy in place. Most likely the ObEx runtime would clear the sensitive data much earlier than the managed code's portion is garbage collected.

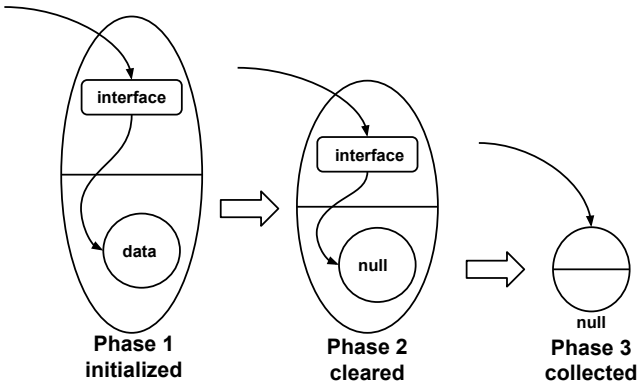


**Figure 2: Life Cycle of ObEx.**

## 4.2 System Architecture

The ObEx system architecture comprises three interacting components, as shown in Figure 3: Programming Interface, Local Runtime, and Distributed Runtime. The Programming Interface provides controlled programmatic access to ObEx objects. The Local Runtime is a per-host native library responsible for safekeeping the sensitive data, its lifecycle and operations. The Distributed Runtime is responsible for transmitting ObEx objects across the network securely, while also maintaining their specified access policies.

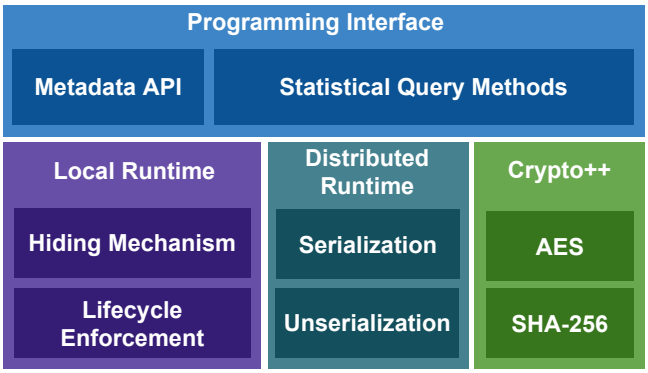


**Figure 3: Structure of ObEx.**

*4.2.1 Programming Interface.* Both the sensitive data's producer and consumers interact with instances of ObEx objects via their programming interface. The producer passes an instance of sensitive data to the constructor of class ObEx alongside with its access policy. Depending on the policy, consumers can invoke different subsets of the API. These subsets expose various query methods that include statistical and comparison operations. The sensitive data remains invisible and inaccessible to consumers.

*API Specifics.* Figure 4 depicts the main structure of the ObEx class, which reifies the access semantics specified by declarative policies. ObEx provides methods for initializing, querying, deep-copying, and statistical calculations. To control the deep-copying and serialization behavior of ObEx objects, the class is declared as `Externalizable`, taking full control of marshaling its instances.[7] The class includes the `private` field `mID`, operation methods, and `native` method declarations. `mID` uniquely identifies ObEx objects, serving as the key that the API methods use when querying sensitive data. To ensure the `mID`'s uniqueness, its value is the sensitive data's digital signature. Conversely, the query methods are simple wrappers around the `native` functions that interact with the local runtime. Specifically, each `private native` function has a corresponding `public` wrapper method.

We designed other classes to support the functionality provided by ObEx. The abstract class `AccessPolicy` serves as the base class for access policies. Figure 4) shows how the API facilitates the implementation of policies via a subclass `CustomPolicy`. The `Methods` `enum` defines the query methods consumers are permitted to invoke on the invisible sensitive data. The data producer is also required to specify the stored sensitive data's type, which are currently confined to built-in Java primitive types and `String`. This provision is necessary to be able to execute all operations on sensitive data within the native layer, without having to pass the sensitive data to the managed layer.

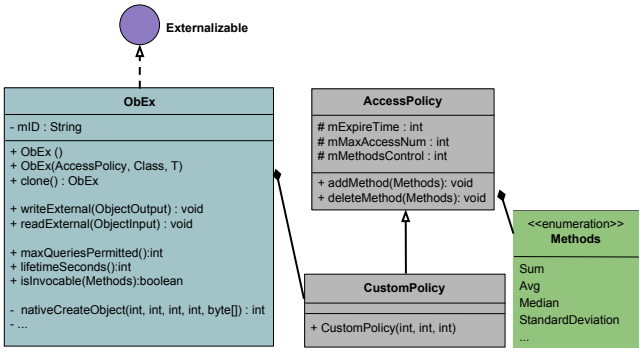


**Figure 4: Class Diagram of ObEx.**

*Metadata API.* The metadata API appears in Figure 5. The metadata provides information about the protection mechanisms of a given ObEx object. The method names are self-documenting. The `getDescription` return a textual representation of the sensitive data represented by a given ObEx. In the presence of multiple sensitive data feeds, this method can be invoked to differentiate between them. `maxQueriesPermitted` returns an ObEx object's maximum number of queries, while `lifetimeSeconds` returns its time-to-expiration. The `isInvocable` method can be used to determine which query methods are permitted.

[7] Notice that custom serialization would not reveal the protected sensitive data, as it is managed in the native layer, which is inaccessible to serialization libraries.

```
1 public native String getDescription()
2 public native int maxQueriesPermitted()
3 public native int lifetimeSeconds()
4 public native boolean isInvocable(Methods enm)
```

**Figure 5: Query Interface for Metadata.**

*Statistical Query Methods.* ObEx supports two types of statistical queries. The first one applies to a single ObEx object, while the second applies to a collection of ObEx objects. Figure 6 shows the query methods that can be invoked on a single ObEx instance. The `equals` and `greater` methods can be invoked to compare some consumer-provided data with the invisible data. For example, an ObEx object can contain a secret verification code. The consumer can invoke a limited number of `equals` method to confirm a user. If the number of invocations or the specified lifetime is exceeded, the actual verification code, the sensitive data in this case, will become inaccessible. Consumers can also learn mathematical boundaries of the hidden data by invoking the `greater` method.

```
1 public native boolean <T> equal(T data)
2 public native boolean <T> greater(T data)
```

**Figure 6: Query Interface for a Single ObEx Object.**

For statistical calculations over multiple objects, ObEx provides a set of statistical methods. These methods are all implemented in the native layer. Our design assumes that all the ObEx objects transferred to the same consumer are homogeneous, containing the same type of data. The data-intensive application in place determines what data it is (e.g., temperature readings, GPS locations, velocity, etc.) These generic methods return the statistical calculations as determined by the sensitive data on which they operate.

```
1 public static native <T> T sum()
2 public static native <T> T average()
3 public static native int count()
4 public static native <T> T median()
5 public static native <T> T[] mode()
6 public static native <T> T stdDeviation()
7 ...
```

**Figure 7: Query Interface for Multiple ObEx Objects.**

Figure 7 shows the query methods that can be invoked on collections of ObEx objects. These methods provide basic statistical functions, including sum, average, count, median, mode, and standard deviation. Some of the functions come in multiple flavors. For example, consumers can both calculate the average of a collection of invisible data, and also can do so with a given value range.

An important property of these queries is that they only include unexpired ObEx objects. The expired objects are dynamically removed from the examined collection. Hence, running the same query in sequence may produce different results. This property can be leveraged to monitor the currently provided and still available sensitive data, protected by ObEx objects.

*4.2.2 Local Runtime.* The ObEx local runtime hides the sensitive data, enforces the data's lifecycle, and executes query operations. Internally, the local runtime organizes the sensitive data by means of two hash tables, which map ids to data and metadata (Figure 8). Data consumers interact with ObEx objects by means of their operations, which are implemented at the native layer. The operations include initialization, cloning, metadata querying, and statistic calculations. Meanwhile, the execution of these operations is governed by the ObEx object's access policy; this policy enforces the object's lifecycle. To support all cryptographic functionality in ObEx, our implementation integrates a third-party C++ library, Crypto++ [8]. For example, this library provides the SHA-256 [19] algorithm, used to compute a digital signature that serves as a unique id for ObEx objects. To sum up, it is the local runtime that keeps the sensitive data invisible, while enabling consumers to query the data, thus preserving user privacy.

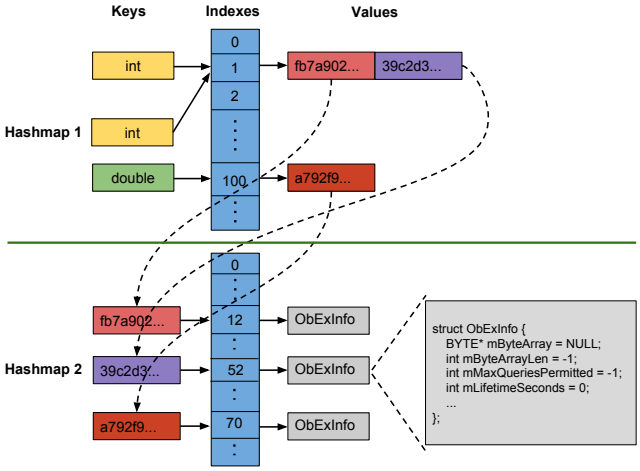


**Figure 8: Data Structure of ObEx (Hash Tables).**

*Privacy Preservation Mechanism.* Figure 9 shows how ObEx preserves user privacy by hiding sensitive data, while enabling consumers to leverage the data's properties. Data producers hand over the sensitive data that needs privacy protection by creating an instance of an ObEx object. The sensitive data is passed as a parameter to the ObEx constructor, with the ObEx runtime converting the data to a binary buffer. Consumers can then request a data feed of ObEx objects from the producer.

In response to receiving such a request, the producer initiates a distributed communication, through which the ObEx runtime securely transfers the sensitive data to the requesting consumer, passing the sensitive data to its respective ObEx local runtime. The local runtime maintains the sensitive data at the system level, with all queries performed by means of `native` JNI methods. The ObEx Java methods are simply wrappers over these `native` methods. To sum up, this process ensures that the sensitive data guarded by ObEx remains invisible at the bytecode level. In other words, the producer's sensitive data cannot be accessed directly by the consumers, thus preserving user privacy.

[8] Crypto++ Library 5.6.5 https://www.cryptopp.com/wiki/Main_Page

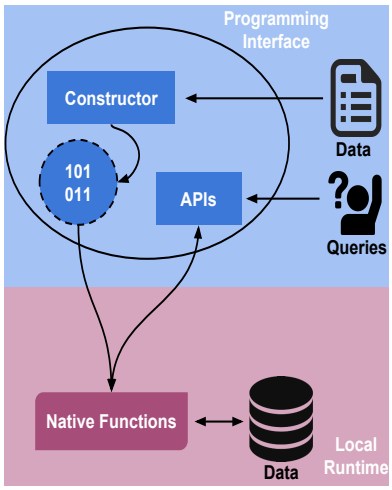


**Figure 9: Privacy Preservation Mechanism.**

*Lifecycle Enforcement.* In addition to hiding sensitive data, the ObEx runtime enforces the data's lifecycle. That is, the sensitive data remains queryable only for a specified time period; after this period has been exceeded, the data becomes inaccessible for any future queries. To support this lifecycle enforcement policy, the ObEx runtime implements two different schemes, depending on whether a given ObEx object guards a single sensitive data item or a collection of them. These schemes are realized as follows.

(1) Figure 10 shows the lifecycle enforcement process for a single sensitive data item. First, a data consumer initiates an operation on an ObEx object (step 1). For example, the consumer wants to check whether the data `equals` some value. The runtime then checks the object's access policy to determine if (a) the object's lifetime has not passed, (b) the operation is permitted, and (c) the max number of operations has not been exceeded (step 2). If any of these three conditions is unmet, the runtime nullifies the sensitive data (step 3). Finally, the runtime returns the operation's result or raises an exception (step 4).

(2) Figure 10 also shows the lifecycle enforcement process for multiple sensitive data items. Different from the single item case, after receiving an invocation (e.g., `average`) (step 1), the runtime checks all the ObEx objects in the collection for the conditions (a), (b), and (c) above (step 2). Then, the objects for which these conditions are unmet have their sensitive data nullified (step 3). Later, the operation's result takes into account only the sensitive data of unexpired objects; if no objects remain accessible, a runtime exception is raised (step 4).

*4.2.3 Distributed Runtime.* The ObEx distributed runtime is responsible for transmitting ObEx objects across the network, while also maintaining their specified access policies. Figure 11 shows the entire data transfer procedure. For the data consumers' perspective, the behavior of ObEx object is similar to a regular Java object. The process has the following steps: (1) serialize the object to a binary stream, (2) transfer it to the non-trusted parties via standard web services over the network, (3) unserialize the binary stream and

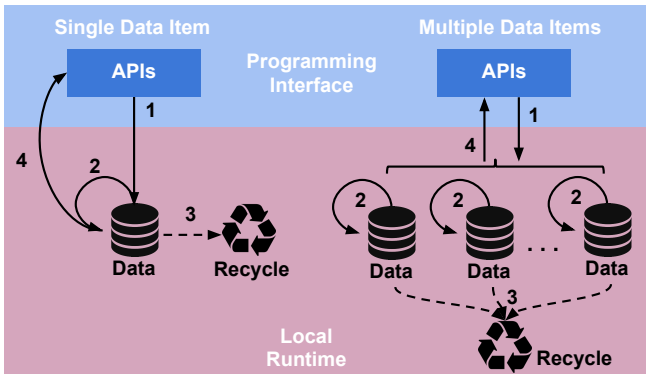


**Figure 10: ObEx Lifecycle Enforcement Process.**

reconstruct the object. However, the ObEx distributed runtime is responsible for serializing / unserializing the objects.

(1) During the serialization, the runtime converts the sensitive data and the metadata into a binary stream. After encrypting it via AES algorithm, it returns the result, which can be integrated with the rest of the serialization process in the managed layer.

(2) During the unserialization, the runtime decrypts the arrived binary stream. Then, it recalculates the sensitive data's digital signature, comparing it with the original object's id, so as to verify the integrity and correctness of the ObEx object. Next, hash tables are set up for efficient lookups. Finally, the runtime restarts the expiration timer, thus making the reconstructed object available for queries.

To sum up, the distributed runtime takes charge of the construction/reconstruction, encryption/decryption, and verification for ObEx objects. These objects can be transferred alongside regular Java objects, while securing their sensitive data and corresponding access policy.

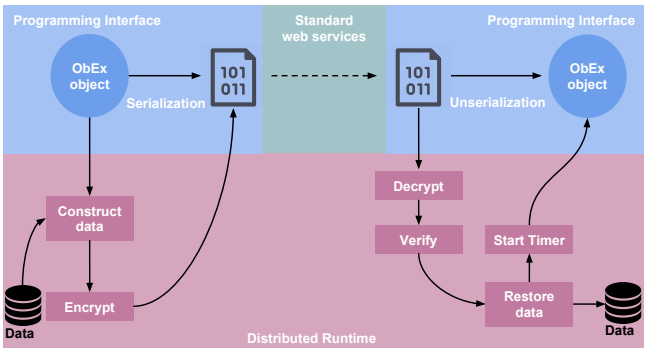


**Figure 11: Data Transfer Procedure.**

## 5 DEVELOPING DATA-INTENSIVE, PRIVACY-PRESERVING APPLICATIONS

To develop an effective application that ensures sensitive data privacy, the data producer must determine (1) how long the sensitive data should remain queryable? (2) how many queries should it permit? (3) which queries should be allowed? Obviously the answers to

these questions are application-specific. We discuss each question, demonstrating the issues involved with a concrete example.

## 5.1 Time-to-Expiration

To define a reasonable lifetime value for sensitive data, a data producer must consider: (1) whether the sensitive data should be observed in real-time, and (2) for how long the access to the data should be granted to consumers. As an example demonstrating the first questions, consider intelligent navigation applications, in which the GPS data should be updated in time to compute current traffic information. In other words, only the latest data is valuable, while the outdated data should be excluded from the statistical calculations. Meanwhile, the frequency of real-time data arrival can determine the lifetime as well. For example, if a health tracker provides heartbeats every 30 seconds, the lifetime of > 90,000 milliseconds would enable calculating the averages of 3 consecutive heartbeats.

As an example demonstrating the second question, consider sending a hidden verification code, which should expire in 30 seconds; or limiting access to the license of commercial software to one hour; or leasing a marketing dataset for 24 hours. Hence, the actual business properties of sensitive data determine the lifetime value that data producers should assign to the data's ObEx object.

## 5.2 Max Number of Accesses

Statistical privacy provides mathematical formulæ to help determine the max number of accesses that preserves user privacy [12]. However, the number of accesses can be determined via business analysis in many cases. We further explicate this issue in section 2.

*Statistical Model.* Recall the motivating use case, in which a person communicating with contacts of similar age is likely to be a local. By knowing the average and standard deviation of a customer's contacts, enterprises can infer whether the customer is likely a local or migrant. If a customer is already determined to be a local, this fact can be leveraged to also learn the customer's age by continuously invoking methods `greater` or `equal`. Next, we describe two different algorithmic approaches that a malicious data consumer can exploit to guess a customer's age, a data item whose privacy we want to preserve.

*(1) Binary Search.* Binary search can find an integer between $0...n$ in $\log_2 n$ comparisons. Assume that a customer's age can range between 1 and 100, while the age's probability distribution is unknown in advance. By using binary search, one can guess the age's value in $\log_2 100 \approx 6.6$ queries. Thus, to reduce the risk of leaking the age information to non-trusted parties, the producer should set the maximum number of permitted `greater` queries to fewer than 6.

*(2) Guessing Entropy.* In probability theory, the guessing process is an "uncertain event" to developers, and the Shannon entropy [18] can be used to determine the uncertainty of information. Christian Cachin [1] proposed a guessing entropy for estimating the expected number of guesses under an optimal guessing strategy. In this paper, we apply this guessing entropy to determine the number of accesses data providers should specify in the access policy of an ObEx object.

Let $\chi$ be a probability distribution, and $x$ be a specific event that can be considered as the correct age in our case. Then, let $X$ be the random variables representing the age guessed by data consumers. Meanwhile, $X$ has $N$ possible values, and the probability of $X$ is $P_X$, so that the elements of $P_X$ will be $p_1, p_2, ..., p_N$.

Therefore, our case could be described based on the mathematical notations above: the age ($x$) has been already stored in the ObEx object. The developer has collected $N$ possible age values ($X_i$) with probabilities $p_1, p_2, ..., p_i, ..., p_N$. Thus, "is $X_i$ equal to $x$?" will be the `equal` querying process of data consumers. In addition, we assume $P_X$ is a monotonically decreasing sequence, which is $p_1 \geq p_2 \geq ... \geq p_{N-1} \geq p_N$. Then, the optimal guessing strategy is to guess the most likely value first, following the sequence of $P_X$. The guessing entropy [1] is:

$$E[G(X)] = \sum_{i=1}^{N} p_i \cdot i \tag{1}$$

The formula calculates the expected number of accesses the developer needs to query an ObEx object to obtain the correct age under the optimal guessing process.

*An Example.* An ObEx object represents the hidden age value as an integer, whose range is 0...100. Assume the local's age is 25, while a consumer has collected a data set including the possible age values of 30, 25, 28, 26 with the respective probabilities of 40%, 30%, 20%, 10%. Based on the guessing entropy, the expected number of `equal` quires is $40\% \cdot 1 + 30\% \cdot 2 + 20\% \cdot 3 + 10\% \cdot 4 = 2$. Therefore, the access policy should set the number of accesses to less than 2 to reduce the risk of leaking the sensitive age data to non-trusted parties.

## 5.3 Permitted Query Methods

To systematically determine which query methods should be permitted on a given ObEx object, data producers should follow the following procedure:

*(1) Inspect Statistical Significance.* The semantics of sensitive data determines which operations can be meaningfully applied to it. For example, it would be absurd to allow clients to only compute the `sum` of observed individual body temperatures or to compute the `average` of GPS locations.

*(2) Consider Privacy Requirements.* Data producers should limit query methods to meet the requirement of data privacy. In our use case, allowing consumers to `sum` a collection of ages is harmless, but for financial data, this query can reveal sensitive information.

*(3) Exclude Composite Methods.* Notice that some statistical calculations can be substituted by a combination of several other operations. For example, one can calculate the `average` of a collection by combining `sum` and `count`. In this case, if the former method is inaccessible, then the latter ones should be excluded as well.

# 6 EVALUATION

We first evaluate various performance characteristics of the reference implementation of ObEx; we then show how ObEx defends

against a deep-copy attack, intended to subvert the ObEx mechanism for protecting sensitive data; finally, we report on the lessons learned from the evaluation.

To assess the performance characteristics of ObEx, we first measure the total time it takes to create, serialize, and statistical query ObEx objects, which encapsulate sensitive data of various sizes. We then measure the total memory consumption at runtime to determine the actual limit on the number of ObEx objects that can be created without exhausting the total JVM memory.

## 6.1 Control Group Choice and Runtime Environment

The ObEx object can be considered efficient if its performance is competitive with other existing APIs provided by Java platform. Thus, we compare the speed of initializing and serializing ObEx objects against three Java APIs, which provide relevant functionalities, including `SealedObject`, `SignedObject`, and `String` [9]. We include `SealedObject` and `SignedObject` because they also provide internal protection mechanisms that improve data privacy. We include `String`, as it is the most commonly used object type with similar features (storing a byte stream). To ensure a fair comparison, we configure the `SealedObject` and `SignedObject` objects with the same encryption algorithm (i.e., `AES` and `SHA-256`) used by ObEx. Table 1 shows the runtime environment we used for all experiments.

**Table 1: Runtime Environment.**

| | |
|---|---|
| OS | ubuntu 16.04 LTS, 64-bit |
| Memory | 11.5 GiB |
| Processor | Intel Core i5-3210M CPU @ 2.50GHz * 4 |
| JDK | OpenJDK 1.8.0_131 |
| JVM | OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode) |

## 6.2 Performance

We compare the respective performance of our subjects in terms of the total runtime and memory consumption. The time consumed by initialization, serialization / unserialization and statistical queries is measured by means of `System.currentTimeMillis`. A Java instrumentation package, running an agent monitoring the memory status at runtime, is used to measure memory consumption.

To determine how the size of sensitive data affects performance, we evaluate a data series of different sizes, ranging from 10 bytes to 1 megabytes. Furthermore, to increase the evaluation's reliability, we repeat each measurement for 1,000 times and average the results.

### 6.2.1 *Time consumption.*

*Instantiation.* Instantiation comprises memory allocation, default value initialization, shared library setup, etc. ObEx invokes a native function to initialize ObEx objects, which allocates variables and data structures, generates the metadata (e.g., current time), calculates the unique object ID, passing it to a field in the managed layer. To measure the time consumed, we record the time before and after creating the object, with Figure 12 showing the results.

As expected, the time consumed increases as sensitive data grows in size. In contrast to the control group, which experiences a sharp spike in execution time as the data reaches 1 megabyte in size, ObEx maintains the average execution time of (11.913ms). Although ObEx takes longer to initialize for small objects than `String` and `SealedObject`, it maintains stable performance characteristics irrespective of the sensitive data's size.

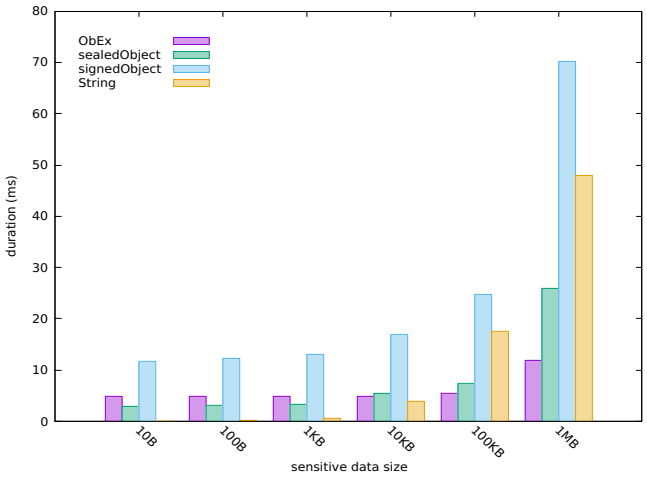


**Figure 12: Performance of Instantiation.**

*Serialization / Unserialization.* ObEx objects are serialized/unserialized when transferred across the network. Normally, an object can be serialized to a binary stream and sent to another device or server. When it arrives to the destination, it is unserialized and reconstructed to an isomorphic object, with the same data fields. When transferring ObEx objects across the network, the runtime first gathers the relevant sensitive data and metadata, and then encrypts and combines them into a byte array as part of serialization. Unserialization reverses the process.

Figure 13 shows the total time consumed by the serialization process. Due to the time taken by assembling and encrypting data, the objects in the control group outperform the ObEx objects. However, for `String` objects, as the contained data increases in size, the time consumed grows rapidly. When the data size reaches 1 megabyte, ObEx (31.32ms) surpasses `String` (49.341ms).

Meanwhile, Figure 14 depicts the time consumed by the unserialization process. ObEx outperform both `SealedObject` and `SignedObject` for small data sizes (from 1B to 10KB). For 1MB, ObEx (16.239ms) only outperforms `String` (26.23ms).

As part of their serialization/unserialization ObEx objects go through expensive operations, such as encryption/decryption, construction/reconstruction, digital signature generation, and verification, thus losing out to the control group, for which these processes are not as involved. ObEx still outperforms `String`. Thus, one can conclude that ObEx shows satisfying performance characteristics as compared to the control group.

[9] Java Platform, Standard Edition 7 API Specification
https://docs.oracle.com/javase/7/docs/api/

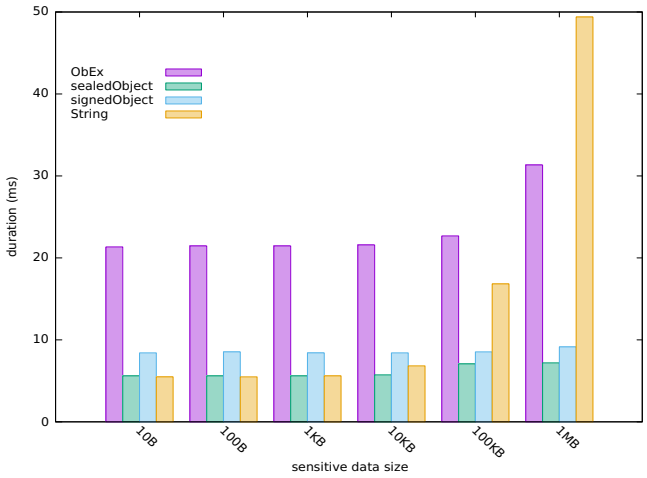


**Figure 13: Performance of Serialization.**

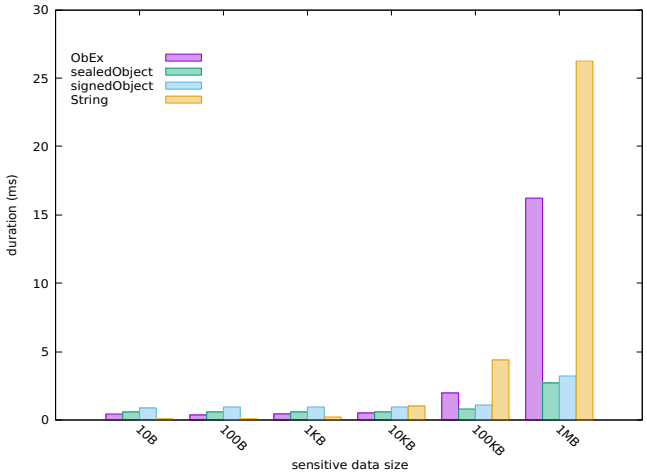


**Figure 14: Performance of Unserialization.**

*6.2.2 Statistical Queries.* Here we measure the total time taken by the statistical calculations a data consumer may want to perform on the ages of a phone user's contacts, including average, count, median, mode, and standard deviation. We first randomly generate age numbers to create a collection of ObEx objects. We then record the time taken to perform each statistical method. Table 2 shows that the time increases linearly with the growth of the number of ObEx objects. Notice that the ObEx runtime first checks if an object may have expired before including its data into a given statistical calculation, thus incurring additional processing time. As future work, we plan to investigate how to parallelize the processing of large collections.

**Table 2: Time Consumption of Statistical Queries.**

| Num | Avg | Count | Median | Mode | Std. |
|---|---|---|---|---|---|
| 10 | ≈ 0ms | ≈ 0ms | ≈ 0ms | 1ms | ≈ 0ms |
| 100 | 4ms | 5ms | 4ms | 5ms | 4ms |
| 1000 | 543ms | 551ms | 553ms | 554ms | 546ms |

## 6.3 Memory Consumption

For data-intensive applications, it is important to ensure that ObEx is memory efficient. Therefore, we first compare the memory consumed by ObEx and the control group, containing data items ranging between 10 bytes and 1 megabytes in size. We investigate the upper limit on the number of ObEx objects that can be created in a single, default configuration JVM.

*Runtime allocation.* To measure memory consumption, we execute an agent JAR file containing an implementation of `premain-class`. By invoking `java.lang.instrumentget.getObjectSize`[10], we can accurately estimate the runtime memory allocated for each object. However, this method can only measure the amount of memory consumed by the specified object. In other words, the fields inherited from superclasses or the actual size of instances in a reference array will be omitted. To address this problem, we sum all object fields by recursively traversing them through Java reflection.

Since ObEx objects run in both the managed and native layers, we measure the memory consumed in both of them. In the managed (i.e., bytecode) layer, the memory allocated for ObEx is fixed to 96 bytes in all different cases of sensitive data sizes (Table 3). The reason is that only the `mID` and several interfaces of wrapper method are stored in the managed layer. In the native layer, ObEx objects consume the amount of memory proportional to the size of the sensitive data they contain. Besides, ObEx objects have metadata attached to them, containing their lifecycle policy, including creation time, the number of accesses, and data type. We sum the memory consumed by all these data items and compare the result with the control group (Table 4).

Table 3 shows that the memory consumed by the ObEx objects is almost the same as that of `SealedObject` and `SignedObject`, which are proportional to the size of the sensitive data they encapsulate. `String` objects allocate two times as much memory as the other subjects for the 1MB data size. These results indicate that ObEx never consumes excessive volumes of memory. More importantly, the ObEx runtime manages the memory allocated for ObEx objects explicitly rather than relying on garbage collection as the control group does.

*Memory Limitation of ObEx Objects.* As compared with Java objects, ObEx objects keep the majority of data in the native layer. In other words, if someone continuously creates an ObEx object with infinite expiration time, the runtime memory will be exhausted. In general, this limitation highly depends on the size of physical memory. We can observe this phenomenon by creating an ObEx object with 1 megabytes sensitive data in an infinite loop, setting the expiration policy to infinity. Table 5 shows the percentage of

[10]Package java.lang.instrument
http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html

**Table 3: Memory Allocation of ObEx.**

| Data Size | Runtime Layer | Managed Layer |
|---|---|---|
| 10 B | 94 B | 96 B |
| 100 B | 184 B | 96 B |
| 1 KB | 1108 B | 96 B |
| 10 KB | 10324 B | 96 B |
| 100 KB | 102484 B | 96 B |
| 1 MB | 1048660 B | 96 B |

**Table 4: Memory Allocation of ObEx and Control Group.**

| Data Size | ObEx | SealedObject | SignedObject | String |
|---|---|---|---|---|
| 10 B | 190 B | 120 B | 272 B | 40 B |
| 100 B | 280 B | 200 B | 360 B | 216 B |
| 1 KB | 1204 B | 1128 B | 1288 B | 1960 B |
| 10 KB | 10420 B | 10344 B | 10504 B | 19616 B |
| 100 KB | 102580 B | 102504 B | 102664 B | 196664 B |
| 1 MB | 1048756 B | 1048680 B | 1048840 B | 2014080 B |

runtime memory consumed by continuously creating ObEx objects. After the number increased near to 10,000, the test environment hangs without responding. This result shows that the maximum possible number of ObEx objects depends on the size of the available physical memory. Our runtime environment, with 11.5 GiB physical memory, can support up to 10,000 objects, each containing 1 megabyte of sensitive data.

**Table 5: Memory Limitation of ObEx Objects.**

| Data Size | Number of Objects | %MEM |
|---|---|---|
| 1 MB | 2000 | 17.6 |
| 1 MB | 4000 | 34.7 |
| 1 MB | 6000 | 51.7 |
| 1 MB | 8000 | 68.0 |
| 1 MB | ≈10000 | N/A |

## 6.4 Object-Level Privacy Threats and Defense

OOP prescribes that objects should encapsulate both data and behavior. Hence, object-level privacy threats try to maliciously gain access to the data by invoking private methods or deep copying the entire object. Attackers can use Java reflection API to perpetrate the first two operations, and can deep-copy objects by serializing them to a memory buffer and reading them back.

ObEx objects are not vulnerable to the reflective access vulnerability, as they keep no sensitive data at the bytecode level. Deep-copying, on the other hand, can create copies that can be forwarded to multiple sites, multiplying the specified access privileges. In this experiment, we emulate this copying attack to check if the ObEx design can defend against it. Figure 15 show how when deep-copying an ObEx object, only the managed layer's content is duplicated, while the sensitive data in the native layer is not. In other words, the copies alias the same sensitive data. The presence of aliases has no effect on the sensitive data's lifecycle.

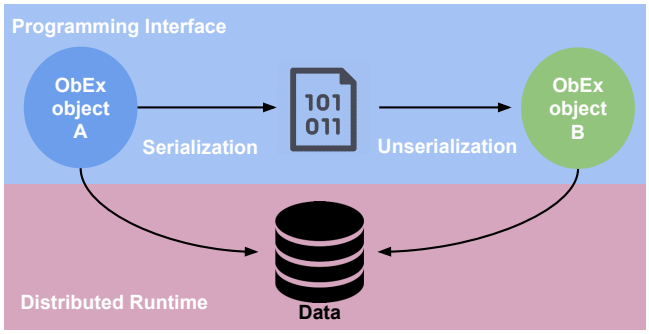


**Figure 15: Process of Deep-Copying ObEx Object.**

## 6.5 Lessons Learned

Despite the privacy-preservation benefits of ObEx demonstrated above, to fully realize its potential, we argue that it should be supported natively by the JVM. This native support would help (1) complicate privilege escalation and library substitution attacks, (2) integrate the ObEx life cycle management with garbage collection, and (3) avoid the dependence on third-party security libraries. We next explain these expected benefits.

*6.5.1 Complicating Privilege Escalation and Library Substitution Attacks.* Because the ObEx local runtime is a shared library loaded by the JVM, privilege escalation (e.g., `root` account in Linux) can directly access the sensitive data in memory. In addition, an attack can replace the ObEx runtime with a malicious version at load time.

Integrating with the JVM can increase the integrity of the ObEx privacy-protection mechanism, as the JVM security mechanism makes it a hard target for attackers[11] [12]. Although the highest privilege attacks still get unrestricted access to the system, the success rate of such attacks is rare for properly administered systems.

*6.5.2 Integrating with JVM Garbage Collection.* After the ObEx runtime clears sensitive data, the memory allocated for its ObEx object should become reclaimable immediately. However, to be able to mark ObEx objects as available for garbage collection requires JVM integration.

*6.5.3 Avoiding Dependencies on Third-Party Security Libraries.* Despite the existence of the JDK package `javax.crypto.*` [13], the ObEx runtime relies on a third-party C++ library (Crypto++), as all sensitive data operations are performed in the native library. Integrating with the JVM would allow using its built-in crypto libraries.

To facilitate the proposed integration, we have started incorporating ObEx into the OpenJDK HotSpot VM. We have already added the programming interface. We are still working on integrating our native library. Appendix provides details of this integration effort.

[11] JVM Specification http://docs.oracle.com/javase/specs/jvms/se7/html/index.html
[12] Secure Computing with Java: Now and the Future http://www.oracle.com/technetwork/java/javaone97-whitepaper-142531.html
[13] javax.crypto.* https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html

# 7 RELATED WORK

ObEx is related to several research area, including differential privacy, authentication, access control schemes, self-destruction mechanisms, and the internal protection in programming languages.

## 7.1 Data Privacy

Data privacy research divides between mathematical approaches and software engineering solutions. In mathematics, Dwork et al. [2] were first to put forward *differential privacy*, a mathematical solution that prevents attackers from maliciously discovering an individual's private information. As a protection mechanism, one can obfuscate the original dataset by generating random noise, while applying the mathematical frameworks of differential privacy to calibrate and measure the impact of the added noise on the statistical operations over the dataset [3]. Zhang et al. [27] proposed how an imperative language can be applied to facilitate the process of verifying differential privacy algorithms.

In software engineering, Gaboardi et al. [8] provided an information sharing interface that enables users, without any differential privacy background, to conveniently generate privacy-preserving datasets that support statistical queries. Meanwhile, Liu et al. [15] presented a programming framework, Oblivm, a domain specific language that enables programmers to create cryptographic programs by using a custom compiler that generates code for secure computation. In addition, Kosba et al. [13] developed "Hawk", a blockchain-based contract system that compiles non-secure "Hawk" programs to those that guarantee the privacy of transactions by employing cryptography. Furthermore, Miller et al. [16] developed a tool that can generate secure protocols, enabling clients to security communicate with a non-trusted server.

## 7.2 Authentication and Access Control

Holford et al. [11] presented a self-defending object (SDO) that can authenticate users while invoking a method. An authentication token is passed as a parameter to the objectfis public methods to be able to examine whether the caller is permitted to invoke a given method. Meanwhile, Venelle et al. [20] provided a Mandatory Access Control (MAC) model for JVM that limits which methods can be invoked and which fields can be accessed. Furthermore, several related works study the Android permission system [5, 6], which can be circumvented to grant illicit access to resources[4], while aiming at improving its effectiveness [21], and identifying the data privacy risks [14, 17].

## 7.3 Self-Destruction

Geambasu et al.[9] developed a self-destruction system called Vanish, which periodically destroys expired access keys, so as to prevent all further access to the encrypted information. Based on a similar concept, Xiong et al. [23] proposed an ABE-based secure document self-destruction (ADS) scheme for the sensitive documents in the cloud. Meanwhile, Yue et al. [25] created a similar self-destruction scheme called SSDD for electronic data. However, Wolchok et al. [22] pointed out that attackers can compromise the Vanish approach by continuously crawling the storage to gain the keys. Although Zeng et al. [26] improved the Vanish approach to avoid the sniffer attacks, their approach was still confined to an external security framework rather than being integrated with a programming language runtime system.

## 7.4 Language Protection Mechanisms

The Java standard API provides an internal protection mechanism for sensitive data via the `SignedObject` and the `SealedObject` mechanisms. The `SignedObject` stores a signature, thus protecting its serializable representation. Without the valid digital signature, the protected object cannot be extracted. The `SealedObject`, on the other hand, encrypts the original object, encapsulating the result with a cryptographic cipher. However, once the original object has been recovered, the sensitive data is no longer protected. Meanwhile, since the implementation of these mechanisms is solely in the Java layer, a native library could be used to intercept and compromise the security mechanisms provided by these objects.

As compared with these works, ObEx relies neither on compiler nor on encryption. The runtime reliably destroys the sensitive data, as prescribed by a declarative access policy. The policies specify the lifetime of sensitive data, the types of queries allowed, and the number of queries permitted. In the meantime, ObEx manages sensitive data entirely in the runtime system layer, thus rendering the data invisible. Hence, ObEx is able to improve data privacy in all managed execution environments, with IoT and mobile applications being particularly promising as an application area.

# 8 CONCLUSIONS AND FUTURE WORK

The advent of mobile, wearable, and IoT devices has generated a deluge of data, much of which has some privacy restrictions. There is an inherent conflict between the end users contributing this data and commercial enterprises. The end users want to keep their sensitive data private, while the enterprise would like to use this data to provide intelligent, context-sensitive services and applications.

In this paper, we argue that innovations in programming technology can help reconcile these two conflicting agendas: sensitive data can be kept private, while enterprises can still derive valuable business intelligence from the data. We show how we designed, implemented, and evaluated ObEx, a novel programming abstraction that keeps sensitive user data invisible, while controlling its lifecyle and querying policy. The ObEx provides programming interfaces to perform statistical computations, so as to enable developers to build intelligent mobile and IoT applications.

We have discussed realistic scenarios and use cases that apply ObEx to preserve data privacy. We have discussed the ObEx design, implementation, and programming model. Our evaluation also shows that ObEx exhibits satisfying performance characteristics. To maximize the positive impact, we advocate that the support of ObEx objects be incorporated into the Java Virtual Machine and exposed via a standard API. To that end, we have motivated why ObEx should be integrated with Java Virtual Machine, as well as discussed (in Appendix) how the ObEx programming APIs can be incorporated into OpenJDK HotSpot VM.

To fully realize the promise of ObEx, we plan to investigate the following questions: (1) To what extent can ObEx reduce the data leakage and privacy attacks? (2) How can we apply ObEx to specific domains? (3) How can we efficiently manage memory and perform the statistical calculations for a large number of ObEx objects?

## REFERENCES


[1] Christian Cachin. 1997. *Entropy measures and unconditional security in cryptography.* Ph.D. Dissertation. Swiss Federal Institute of Technology in Zurich.

[2] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*. Springer, 1–19.

[3] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*. Springer, 265–284.

[4] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 627–638.

[5] Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*. 7–7.

[6] 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 SOUPS*. ACM, 3.

[7] Ira R Forman and Nate Forman. 2004. Java Reflection in Action (In Action series). (2004).

[8] Marco Gaboardi, James Honaker, Gary King, Kobbi Nissim, Jonathan Ullman, Salil Vadhan, and Jack Murtagh. 2016. PSI (): a Private data Sharing Interface. In *Theory and Practice of Differential Privacy*. New York, NY. https://arxiv.org/abs/1609.04340

[9] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. 2009. Vanish: Increasing Data Privacy with Self-Destructing Data.. In *USENIX Security Symposium*. 299–316.

[10] Li Gong and Gary Ellison. 2003. *Inside Java (TM) 2 Platform Security: Architecture, API Design, and Implementation.* Pearson Education.

[11] John W Holford, William J Caelli, and Anthony W Rhodes. 2004. Using self-defending objects to develop security aware applications in Javaffl. In *Proceedings of the 27th Australasian conference on Computer science-Volume 26*. Australian Computer Society, Inc., 341–349.

[12] Daniel Kifer and Bing-Rong Lin. 2012. An axiomatic view of statistical privacy and utility. *Journal of Privacy and Confidentiality* 4, 1 (2012), 2.

[13] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 839–858.

[14] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 280–291.

[15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 359–376.

[16] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated data structures, generically. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 411–423.

[17] Majda Moussa, Massimiliano Di Penta, Giuliano Antoniol, and Giovanni Beltrame. 2017. ACCUSE: helping users to minimize Android app privacy concerns. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 144–148.

[18] Claude Elwood Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review* 5, 1 (2001), 3–55.

[19] Secure Hash Standard. 2002. FIPS PUB 180-2. *National Institute of Standards and Technology* (2002).

[20] Benjamin Venelle, Jérémy Briffaut, Laurent Clévy, and Christian Toinard. 2013. Security enhanced java: Mandatory access control for the java virtual machine. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*. IEEE, 1–7.

[21] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. 2014. Compac: Enforce component-level access control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, 25–36.

[22] Scott Wolchok, Owen S Hofmann, Nadia Heninger, Edward W Felten, J Alex Halderman, Christopher J Rossbach, Brent Waters, and Emmett Witchel. 2010. Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs.. In *NDSS*.

[23] Jinbo Xiong, Zhiqiang Yao, Jianfeng Ma, Ximeng Liu, and Qi Li. 2013. A secure document self-destruction scheme: an ABE approach. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 59–64.

[24] Yang Yang, Chenhao Tan, Zongtao Liu, Fei Wu, and Yueting Zhuang. 2017. Urban Dreams of Migrants: A Case Study of Migrant Integration in Shanghai. *arXiv preprint arXiv:1706.00682* (2017).

[25] Fengshun Yue, Guojun Wang, and Qin Liu. 2010. A secure self-destructing scheme for electronic data. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*. IEEE, 651–658.

[26] Lingfang Zeng, Zhan Shi, Shengjie Xu, and Dan Feng. 2010. Safevanish: An improved data self-destruction for protecting data privacy. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 521–528.

[27] Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 888–901.


# Appendices

## A INTEGRATING THE OBEX PROGRAMMING INTERFACE INTO OPENJDK HOTSPOT VM

(I.) Download `OpenJDK` source code

(II. ) Build `OpenJDK`

(III. ) Integrate ObEx Programming Interface

Steps:

(1) Put ObEx programming interface (.java file) into the `classess` folder [14].

(2) Build the source code for generating header files that are needed for native functions [15].

(3) Match native functions with the header file (may need to modify the original native code).

(4) Add the native functions to the `mapfile-vers` file [16].

(5) Rebuild the `OpenJDK`.

(6) Test

- Use the `java`,`javac` command built by step (5) [17].
- Create tests which use new API and its native functions.
- Compile and run the tests.

[14] Folder: ../jdk8u/jdk/src/share/classes/java/security

[15] Folder: ../jdk8u/build/linux-x86_64-normal-server-release/jdk/gensrc_headers

[16] Folder: ../jdk8u/jdk/make/mapfiles/libjava

[17] Folder: ../jdk8u/build/linux-x86_64-normal-server-release/images/j2sdk-image/bin