EGEON: Software-Defined Data Protection for Object Storage

Raul Saiz-Laudo, Marc Sánchez-Artigas

Computer Science and Maths, Universitat Rovira i Virgili

Tarragona, Catalonia, Spain

Abstract—With the growth in popularity of cloud computing, object storage systems (e.g., Amazon S3, OpenStack Swift, Ceph) have gained momentum for their relatively low per-GB costs and high availability. However, as increasingly more sensitive data is being accrued, the need to natively integrate privacy controls into the storage is growing in relevance. Today, due to the poor object storage interface, privacy controls are enforced by data curators with full access to data in the clear. This motivates the need for a new approach to data privacy that can provide strong assurance and control to data owners. To fulfill this need, this paper presents EGEON, a novel software-defined data protection framework for object storage. EGEON enables users to declaratively set privacy policies on how their data can be shared. In the privacy policies, the users can build complex data protection services through the composition of data transformations, which are invoked inline by EGEON upon a read request. As a result, data owners can trivially display multiple views from the same data piece, and modify these views by only updating the policies. And all without restructuring the internals of the underlying object storage system. The EGEON prototype has been built atop OpenStack Swift. Evaluation results shows promise in developing data protection services with little overhead directly into the object store. Further, depending on the amount of data filtered out in the transformed views, end-to-end latency can be low due to the savings in network communication.

Index Terms—object storage, software-defined, serverless, data privacy

I. INTRODUCTION

With the rapid growth in popularity of Cloud services, object storage systems (e.g., Amazon S3, IBM COS or OpenStack Swift [1]) have gained momentum. These storage systems offer consolidated storage at scale, with high degrees of availability and bandwidth at low cost. Proof of that is the recent trend of serverless computing. Due to the high difficulty of function-to-function communication[1], many serverless systems use object storage for passing data between functions [2]–[6], which has revived the interest in this type of object storage.

Although very useful for cloud applications, object storage systems offer a small number of options to keep sensitive data safe. Few (or no) efforts have been realized on security issues such as data confidentiality, data integrity, or access control, to mention a few. For instance, online storage services such as Amazon S3, or IBM COS, only provide server-side encryption for protecting objects at rest [7], [8]. Similar words can be said for the access control of individual objects[2], which is currently either realized via simple object ACLs (Access Control Lists) as in S3 [7], or not possible at all as in OpenStack Swift [9].

This poor interface is insufficient for many applications. For instance, it does not enable in-place queries on encrypted data,

[1]Some works have shown that cloud functions can communicate directly using NAT (Network Address Translation) traversal techniques. However, direct communication between functions is not supported by cloud providers.

[2]In general terms, cloud object stores enforce access at the container level rather than at the object level.

transparent and secure data sharing, and access control based on the content of an object. But also, it is very difficult to make it evolve to meet the changing needs of applications and withstand the test of time. In practice, most of these object storage systems leave no other choice to modifying the system internals to incorporate new security mechanisms at the object level. This requires a deep knowledge of the system, extreme care when modifying critical software that took years of code-hardening to trust, and significant cost and time (see, for instance, [10], where it is described the "daunting" task of deploying new erasure coding solutions in object stores such as OpenStack Swift [1] and Ceph [11]).

Rather than relying on object storage systems to change, we advocate in this paper to "work around" the traditionally rigid object storage APIs by embracing a *software-defined* approach. Similarly to software-defined networking (SDN), we argue that the separation of the "control logic" from the "data protection logic" can give the needed flexibility and ease of use to enable users, programmers and sysadmins to custom-fit access control and object protection. To give an example, pretend that upon certain conditions, parts of an encrypted object need to be re-encrypted to share it with the mobile users of the application. Further, these conditions may depend on the contents of the object itself (e.g., on sensitive data such as sexual orientation), which must always remain confidential from the server. Simply put, what we pursue is to offer users the ability to succinctly express this behavior at the object storage level, and enforce it by calling the corresponding re-encryption modules.

Nevertheless, a software-defined solution to enhance object storage data protection requires solving several issues at two levels:

- At the control plane, by enabling the composition of per-object protection services. These compositions should be expressed concisely and in a manner agnostic to the data protection code and the remaining storage stack.
- At the data plane, by making it truly programmable. To put it baldly, the data plane should not only allow to plug-in new protection logic in the critical I/O path, but to run it safely. In addition, it should enable the re-usability of the protection capabilities, so that users can compose new data protection controls.

In this research work, we present EGEON, a novel software-defined data protection framework for object storage. EGEON exports a scripting API to define privacy policies for protecting objects. These policies enable data owners to declaratively set complex data protection services through the composition of user-defined transformations run in a serverless fashion. These functions represent the elementary processing units in EGEON, and may be re-used and linked together to implement complex privacy policies. The major feature of these transformations is

that they are executed "inline" by EGEON upon a standard `Get` request. In this way, users can trivially display multiple views from the same object, and modify these views by updating the policies. If some functionality to implement a view is missing, EGEON provides a simple API to deploy new transformations and customize the access to data objects.

In this sense, one of the primary contributions of EGEON is the ability to provide privacy-compliant transformed views of the underlying data on the fly. Perception of privacy can vary broadly across applications. As an example, a dataset created by a hospital may include personally identifiable information (PII) that is not needed when it is processed by a data analytics engine. Nonetheless, if the same dataset is accessed by medical personnel, a richer view of it should be given. Thus, a practical system needs to support a range of privacy preferences.

By adopting a software-defined storage architecture, EGEON allows users to express their privacy preferences as policies in the control plane and produce views conforming to the policies by running transformations in the data plane. In this work, we focus on (cryptographic) transformations that process data as streams, that is, as data is being retrieved from object storage nodes. Consequently, the first bytes of transformed views are received as soon as possible, which permits EGEON to scale to arbitrary object sizes without important penalty on end-to-end latency.

The EGEON prototype we present in this research has been implemented atop OpenStack Swift [1]. We took Swift because it is open source and a production quality system. Its sizable developer community ensures that our new properties are built on code that is robust and that will be soon evaluated. It must be noticed that the design concepts underpinning EGEON are generic and could be easily ported to other storage substrates. For example, object classes allow to extend Ceph by loading custom code into Object Storage Daemons (OSDs), which can be run from within a `librados` application [12]. Thus, with some effort, it would be possible to leverage object classes to implement transformed views of data.

Our performance evaluation of EGEON shows promise in developing data protection services with low overhead directly into the object storage. For instance, the overhead of a `NOOP` policy, where a storage function simpy echoes the input data, is of around 9 ms. Also, depending on the amount of protected data filtered out in the transformed views, end-to-end latency can be even lower with EGEON due to the savings in network communication (up to 72.1x for a 4G mobile use case).

II. DESIGN

EGEON is a software-defined data protection framework that augments object storage with composable security services to enforce users' privacy preferences over protected data. These services are built up as pipelines of serverless functions. Fig. 1 illustrates an overview of EGEON's architecture. Our objective is to enable authorized users or applications to access protected data without violating the privacy policies of data owners. We have designed EGEON to make it easy the leverage of state-of-the-art privacy solutions (such as content-level access control, homomorphic encryption, encrypted keyword search, ...) while preserving the normal data flow in the consuming applications. Concretely, we achieve this by introducing a logical separation between the *privacy plane*, where data owners set their privacy preferences, and the *data plane*, where the creation of privacy-compliant transformed views happens. This separation allows

Fig. 1: EGEON's architecture.

for heterogeneous policies atop the same data without having to modify the system internals to enforce advanced policies at the object level.

EGEON's architecture consists of the following components:

Privacy Plane. The privacy plane in EGEON corresponds to the control plane of a software-defined architecture [13], [14], but specialized for data protection. In practice, this means that EGEON provides its own script language to assist data owner in composing data protection services from elementary serverless functions in the *data plane*. The textual, JSON-based language supports conditions and compositions to build up inline privacy transformers (e.g., see Listing 39). Moreover, EGEON offers an API to allow data owners to manage the life-cycle of their data protection policies. For performance reasons, once uploaded, the policies are automatically compiled into Java bytecode and stored in the *Metadata Service*. For fast access, this service has been built on top of Redis [15], an in-memory, low-latency key value store.

Data Plane. In EGEON, the data plane has a critical role. The data plane is responsible for generating the privacy-compliant data views. And hence, it must be extensible to accommodate new functionality that enables privacy transformations on data. Particularly, in this realization of EGEON, we have focused on inline privacy transformations as data is retrieved from storage nodes HDDs. As mandated by the policies in the privacy plane, privacy transformations are constructed from pipelines of user-defined functions executed as serverless functions by EGEON. Namely, a user integrating a new transformation only needs to contribute the logic. Resource allocation and execution of the chain of transformations is automatically handled by EGEON, bringing a true serverless experience to users.

To minimize execution overhead, since many cryptographic operations are CPU-intensive, EGEON abides by the principles of reactive programming and runs a transformation only when is strictly needed, instead of continuously on the data streams. More concretely, EGEON extends the observer pattern [16] and execute a certain transformation in the pipeline when an event

occurs [17]. Consequently, EGEON better utilizes the available resources in the storage servers by balancing the load across the chain of transformations. To better understand this, pretend that a user wants to compute the average salary of employees in a department X. Now suppose that all the employee records have been saved in a single JSON document with all the salary values homomorphically encrypted. Thanks to EGEON reactive core, the transformation to average the salary will only be run when the event "employee of department X" comes through the data stream, thereby saving CPU resources.

A. Threat Model

Specifically, EGEON enforces user's privacy preferences via function composition. That is, users are ensured that their data is transformed as it goes through a pipeline of transformation functions before it is released to applications. In the meantime, the original data remains end-to-end encrypted.

We assume an *honest-but-curious* [18] storage servers, i.e., the server performs the computations correctly but will analyze all observed data to learn as much information as possible. We also presume the existence of an identity service (IDS) such as OpenStack Keystone for user authentication. We leverage this service for authentication of the storage functions. An IDS is a standard requirement in multi-user systems and can even be a trustworthy external entity.

Consumers of shared data are semi-trusted, in that they do not collude with the servers to leak the data or keys. This is a reasonable assumption for groups of data consumers that are acquainted with each other. Further, EGEON assumes that the applications behave correctly and do not hand out user keys to malicious parties. Finally, we assume state-of-the-art security mechanisms to be in place for user devices, and that all parties communicate over secure channels.

In this setting, EGEON enforces *data confidentiality*, making sure that the adversary learns nothing about the data streams, except what can be learned from the transformed views.

Robustness. While EGEON is able to handle various types of failures in practice, provable robustness against misconfigured, or even malicious privacy policies, and data producers is out of scope for this research. A malicious user sending corrupted tokens cannot compromise privacy but could alter the output of a transformed view.

B. Privacy Plane

In the privacy plane, EGEON provides the capabilities for data owner to set their privacy preferences —i.e., user-centric privacy—, and what transformations will be required to apply to enforce a privacy policy. These transformations are specified by their unique name, and their existence is verified when the privacy policy is to be compiled. A privacy policy applies to a single object. In this paper, we do not consider the question of how to set privacy policies for group of objects and how they should look like. This question has been left for future work.

In EGEON, targeted data objects in the privacy policies are specified by its full resource path. Following OpenStack Swift specs [1], the access path to an object is structured into three parts: /account/container/object. As an example, for the rose.jpg object in the images container in the 1234 account, the resource path is: /1234/images/rose.jpg.

Data owners can translate their preferences over an object to a set of transformations by mapping them in a JSON-based schema language. In addition to some meta-information (e.g.,

policy identifier), this schema permits data owners to formalize conditions at the policy level using a rich set of operators such as "StringLike", "NumericLessThan", etc. Importantly, the language enables data owners to build date expressions using operators like "DateNotEquals", which makes it possible to express temporal restrictions. For instance, pretend that a data owner wishes to prevent that a document can be accessed on weekends. She could indicate this through the date expression: "DateNotEquals" : {"Day" : ["Sat", "Sun"]}.

More interestingly, this schema allows composing complex data protection transformations from elementary UDFs. This can be achieved by adding each individual UDF as a step in the transformation pipeline defined in the JSON object "Action". This object contains two name-value pairs: "StartAt", which indicates the first transformation in the pipeline, and "Steps", which is another JSON object that specifies the transformation UDFs along with their input parameters. Since transformations run only when the corresponding events come through the data stream, this schema allows data owners to specify the observed events for each transformation to execute. To do so, there exists a field named "EventType" to signal the event to be observed by a particular transformation UDF. A typical "EventType" block looks like this:

```
{
    "Type": "<type_of_event>",
    "Input": [<parameter_block>, ...],
}
```

where the "Type" field specifies the type of an event (e.g., an XPath [19] event) and the array "Input" lists the parameters that are required for this type of event. For instance, if a data owner wanted to apply a transformation UDF over all salary elements of an XML document, she could do so by setting an XPath event as follows:

```
{
    "Type": "XPathEvent",
    "Input": [{"Predicate": "//salary"}],
}
```

Similarly, a transformation UDF block is defined as follows:

```
{
    "Id": "<identifier_of_UDF>",
    "EventType": "<event_block>",
    "Input": [<parameter_block>, ...],
    "Next": (<identifier_of_UDF>|"End")
}
```

The "Id" field is a string which uniquely identifies the UDF, while the "Input" field permits to specify the parameters for the transformation UDF (e.g., the targeted security level of a cryptosystem). Finally, the "Next" field indicates the next step to follow in the pipeline, or "End" to indicate the end of the chain of transformations.

An example of a real policy can be found in Listing 1. This policy provides transformed views over a JSON file containing employee records of the format:

```
{
    "employee": {
        "name": "Alice",
        "identification": {
            "SSN": "32456677"
        },
        "salary": 50000
    }
    ...
}
```

As a first transformation in the chain, this policy uses content-level access control [9] (CLAC). Very succinctly, CLAC works as follows. It assigns targeted JSON elements an object label (`olabel`) and each user a user-label (`ulabel`). Then, it allows to define rules in the form of (`ulabel`, `olabel`), which means that the JSON elements labeled with any of the `olabels` are allowed to be read by the users labeled with the corresponding `ulabels`. To indicate what JSON items to protect, CLAC uses JSONPath in this case.

In this policy, we assume two types of users: treasurers with user label "`treasurer`" and regular users with label "`user`". We protect the salaries with the object label "`sensitive`" and specify the single rule ("`treasurer`", "`sensitive`"), which means that only treasurers will have access to the salaries. The field "`salary`" is identified using the JSONPath expression: '`$.employee.salary`' (Listing 1, line 14).

The second transformation applies homomorphic encryption on the field "`salary`" to prevent the servers from learning the employee salaries [20], while computing the average salary of employees (Listing 1, lines 20-30).

As a final transformation in the chain, the policy uses proxy re-encryption [20] to convert the homomorphically encrypted average salary to a ciphertext under the receiver's key. Thus, the transformed view can be decrypted by the receiver, without the data owner having to share her private key nor performing any encryption for the receiver on her personal device.

To wrap up, this policy will generate two transformed views of the same data. For regular users, it will only be executed the first step (lines 10-19): the CLAC transformation. Due to lack of permissions, the CLAC module will eliminate the encrypted "`salary`" field, and output a transformed view with the rest of information. For a treasurer, it will output the same view as any regular user (without individual salaries), but enriched with the average salary encrypted under her public key thanks to proxy re-encryption (Listing 1, lines 31-37).

We want to note that the different transformations UDF only execute when the corresponding JSONPath events come along. To wit, proxy re-encryption will only be run one time, after the JSON field named "`average_salary`" is added at the end of the response by the second transformation in the chain.

As a final word, this example clearly displays how EGEON is capable of performing real-time privacy transformations atop the same data object for a variety of application scenarios, thus enhancing the rigid interface of object storage systems.

C. Data Plane

The focus of EGEON is on inline privacy transformations, where data streams are "observables" and user-defined privacy transformations are "observers" subscribed to the data streams. As as soon as an event is observed, it will be delivered to the subscribed observers. If there are no events on the data stream then the original data stream is pushed back to the user. In this sense, a privacy transformation is nothing but a function taking an observable as input and returning another observable as its output. This design has the advantage that transformations can be chained together to generate complex data views compliant with the policies in the privacy plane.

Runtime. Currently, EGEON's runtime is Java-based. Thus, the chain of transformations is run within a Java Virtual Machine (JVM) wrapped within a Docker container to guarantee a high level of isolation between two different data transformations.

```
{                                                          1
 "Id": "employee.policy",                                 2
 "Object": "v1/{account}/{container}/employees.json",     3
 "Condition": {                                            4
    "DateNotEquals": { "Day": ["Sat","Sun"] }             5
 },                                                        6
 "Action": {                                               7
  "StartAt": "Step1",                                      8
  "Steps": {                                               9
    "Step1": {                                             10
     "Id": "CLAC",                                         11
     "EventType": {                                        12
        "Type":  "JSONPathMarkerEvent",                    13
        "Input": [{"Predicate":"$.employee.salary",        14
           "olabel": "sensitive" }]                        15
     },
     "Input": [{"ulabel": "treasurer",                     16
            "olabel": "sensitive" }],                      17
     "Next": "Step2"                                       18
    },                                                     19
    "Step2": {                                             20
     "Id": "SUM",                                          21
     "EventType": {                                        22
        "Type":  "JSONPathEvent",                          23
        "Input": [{"Predicate":"$.employee.salary"}]       24
     },                                                    25
     "Input": [{"average": true},                          26
            {"keyOwner": "meta://Alice/keys/hom",          27
            {"output":"$.average_salary"}],                28
     "Next": "Step3"                                       29
    },                                                     30
    "Step3": {                                             31
     "Id": "PRE",                                          32
     "EventType": {                                        33
        "Type":  "JSONPathEvent",                          34
        "Input": [{"Predicate":"$.average_salary"}]        35
     },                                                    36
     "Next": "End"                                         37
}}}                                                        38
```

Listing 1: A sample policy to process employee records.

Policy Enforcement. Upon a new `Get` request, EGEON starts up a thread inside the JVM to perform three tasks. We refer to this thread as the "master thread". The three tasks in order of execution are:

1) *Policy loading*, where the master thread loads the policy into memory and evaluates its conditions clauses.
2) *Observable setting*, where the master thread opens a data stream to the target object, namely, the observable, if the policy conditions are fulfilled. To this aim, it creates an instance of the appropriate subclass of the abstract class `StreamBuilder` that the EGEON's engine leverages to start parsing the object. Subclasses are required since the specific logic to parse the data stream and generate the events depends on the type of file. Specifically, EGEON picks up the proper `StreamBuilder` subclass based on the object extension (e.g., ".`json`" for JSON documents) using *factory* methods.
3) *Transformations setting*, where the master thread makes an instance of each transformation UDF in the pipeline and chains them together. The `StreamBuilder` subclass generates events as the data is being parsed, and notifies the subscribed transformation UDFs in the same order as dictated in the privacy policy. To wit, in Listing 39, upon the JSONPath event '`$.employee.salary`', EGEON will execute first the CLAC transformation, followed by the SUM transformation.

In EGEON, we assume that an observable can only handle one event at a time. We adopted this design to minimize compute resources at the storage layer, so that both the data stream and all its transformations operate in the same thread, in our case, the master thread. Nevertheless, this can be easily changed by switching to a different thread and integrating some additional

logic for coordinating the threads.

Since each transformation UDF acts as an observer, another responsibility of the master thread is to subscribe each privacy transformation to the events specified in the policy. To this aim, it invokes the method `install(Event event, UDF observer)` in the `StreamBuilder` class, where the parameter `observer` is the transformation UDF bound to the `event`.

Extensibility. At the time of this writing, EGEON implements three types of data sources: XML, JSON and CSV documents, a number of events including XPath and JsonPath expressions, CSV field and records, etc., and multiple transformation UDFs (see §II-D for further details). However, EGEON is extensible, and new events, observables and observers can be incorporated by extending the abstract classes `Event`, `StreamBuilder` and `UDF`, respectively.

Due to space constraints, we only show here an example of a transformation UDF to perform summations on ciphertexts [20] to see how easy it is to code a transformation UDF (Listing 2).

```
1   package com.urv.egeon.function;
2
3   import com.urv.egeon.runtime.api.crypto.Homomorphic;
4   import com.urv.egeon.runtime.api.parser.UDF.ContextUDF;
5   import com.urv.egeon.runtime.api.parser.UDF.UDF;
6   import com.urv.egeon.runtime.api.parser.event.Event;
7
8   public class Sum extends UDF {
9       private Homomorphic accum = new Homomorphic();
10
11      @Override
12      public Event update(Event e, ContextUDF ctx) {
13          try {
14              this.accum.add((String) e.getValue());
15          } catch (Exception ex) {
16              // first execution of the UDF
17              this.accum.setKeys((String)
                      ctx.getParameter("keyOwner"));
18              this.accum.setCipher(this.accum.fromSerial((String)
                      e.getValue()));
19          } finally {
20              return e;
21          }
22      }
23      @Override
24      public Object complete(ContextUDF ctx) { ... }
25  }
```

Listing 2: A transformation UDF to perform summations on ciphertexts.

As shown in Listing 2, a UDF has two methods: the method `update`, which is invoked every time a new subscribed event is emitted, and the method `complete`, which is called when the data stream is finalized (empty). The `update` method has two arguments of type `Event` and `ContextUDF`. The first argument encapsulates the details of the emitted event. For instance, for the homomorphic summation UDF of Listing 2, this may mean a JSONPath event, alongside the value of the selected field by the JSONPath expression (e.g., an encrypted salary value), and accessible via the method `getValue` (Listing 2, line 14). The `ContextUDF` encapsulates the access to the request metadata, such as HTTP headers and cryptographic keys and tokens sent out by the client, the object's metadata, and specific parameters required for the UDF to work, such as the data owner's public key to operate on the encrypted values (Listing 2, line 17). All this information is automatically made available by EGEON to the transformation UDF. This includes the input parameters set in the privacy policy (e.g., Listing 1, lines 26-28).

It is worth to mention here that the input arguments whose values have the format of "`meta://key`" are automatically downloaded from the Metadata Service using the key "key". In this way, a data owner can change the input parameters (e.g.,

her homomorphic public key) without having to re-compile the policy. An example of this can be found in Listing 1, line 27, for the argument "`keyOwner`", whose value is retrieved from the Metadata Service behind the scenes, and made accessible to the summation UDF via the context (Listing 2, line 17).

The purpose of the method `complete` is to provide a hook for developers to perform final computations and append their result at the end of the data stream. For instance, this could be useful to compute the average over encrypted data and append the result as new item at the end of a JSON document.

D. Data Transformation UDFs

EGEON comes up with a library of reusable functions (UDFs) for inline data protection. While some of these functions apply cryptographic transformations, others act on raw data, e.g., by filtering out protected parts of a JSON object to non-privileged users [9]. All functions are composable to provide complex transformed data views. These are the following:

Homomorphic encryption (HOM). HOM is a cryptosystem (typically, IND-CPA secure) that allows the server to perform computations directly on encrypted data, the final result being decrypted by the user devices. For general operations, HOM is prohibitively slow. However, it is efficient for summation. To support summation, along with proxy re-encryption (PRE), we adopted the homomorphic cryptosystem of [20]. We chose this scheme, because it allows secure data sharing and is tailored to mobile platforms and constrained IoT devices. Concretely, we implemented two UDFs:

- Summation (SUM): This UDF supports the summation of ciphertexts, such that the result is equal to the addition of the plaintext values: $\text{Enc}(m_1) \cdot \text{Enc}(m_2) = \text{Enc}(m_1 + m_2)$.
- Re-Encryption (PRE). Succinctly, PRE allows the storage servers to convert ciphertexts under the data owner's key to ciphertexts under authorized users keys without leaking the plaintext. Therefore, the data owner can securely share data with other users, i.e., without sharing her private key nor performing any encryption for them on her personal device. To do so, the data owner d solely needs to issue a re-encryption token for a user u based on his public key pk_u as the $\text{Token}_{d \to u}$. With this token, the PRE UDF can automatically re-encrypt data on behalf of the data owner without her intervention. Further, the re-encryption tokens are unidirectional and non-transitive.

For both UDFs, we assume integers of ≤ 32 bits with 128-bits of security. The implementation makes use of the optimal Ate pairing [21] over Barreto-Naehrigopera elliptic curve [22], and also applies the Chine Remainder Theorem (CRT) to optimize decryption [20]. For all this cryptographic processing, we use the RELIC toolkit [23].

Keyword search (SEARCH). To allow keyword searches (as the SQL "ILIKE" keyword), we make use of a cryptographic scheme for keyword searches on encrypted text [24]. As above, we have chosen this scheme because it is multi-user. To put it baldly, before storing an object, the data owner first selects the users with whom she wants to share her data and then encrypts it with their public keys. To search for keywords in the shared object, a user makes a trapdoor Trap_W for the keyword set W using his private key, and then sends it to the server. The server runs the SEARCH UDF, which takes the public key of the user, the trapdoor Trap_W and the encrypted text, and returns "yes"

if contains W or "no" otherwise. As expected, this scheme is proved secure against chosen keyword attacks (IND-CKA).

One major advantage of this scheme is its short ciphertext size. Concretely, for n users, it requires $(n + \ell + 1) \cdot L1$ bits, where ℓ is the number of keywords and $L1$ is the bit length of the underlying finite field $\mathbb{F}_q$, which is much smaller than the deployment of n separate instances of a public key searchable encryption scheme, one for each user. For the implementation of this scheme, we use the "SS512" elliptic curve, a symmetric curve with a 512-bit base field, which provides a security level of 80-bits, from the jPBC library [25].

Content-level access control (CLAC). As an example of a non-cryptographic function, we decided to implement content-level access control [9]. The central idea of CLAC is to enforce access control at the content level to restrict who reads which part of a document. To give a concrete example, consider that a hospital stores its patient records as big JSON object. These records should be accessed differently by different personnel. For example, a "doctor" could see health information from her patients, while a "receptionist" should only view basic profile information about the patients. With CLAC, a data owner can define content-level policies to censor access to parts of a data object. Remember that in object storage systems such as Swift or Amazon S3, once an object is made accessible to someone, she retrieves the full content of the object. So, there is no way to hide out sensitive information to that user.

Our implementation of CLAC overcomes this limitation. As introduced in §II-B, CLAC borrows the LaBAC model (Label Based Access Control Model) [9]. Either be an XML element, a JSON element, or a CSV column, an object label (olabel) is assigned on the targeted item. Similarly, each authorized user is given a user label (ulabel). Then, the LaBAC model works by specifying tuples of rules in the form of (ulabel, olabel), which tell that only the users labeled with ulabel can access the items labeled with that olabel. For instance, if only users with the user label "manager" were authorized to access items labeled with "restricted", a data owner should have to set the rule ("manager", "restricted") to effectively formalize access control. As in the LaBAC model, our implementation admits hierarchies of both ulabels and olabels to rank users and objects.

One interesting side effect of our reactive data plane is that our CLAC implementation is generic, and not tied to a specific file type. What changes is the mechanism to identify the items, which depends on the object type. That is, for XML, an XPath expression should be used to indicate that a certain element has "restricted" access, while for a JSON document, a JSON predicate should be used in its stead. We have abstracted this coupling by the definition of specific marker events as shown in Listing 1, lines 22-25.

III. IMPLEMENTATION

We have constructed EGEON by extending Zion [26], a data-driven serverless computing middleware for object storage. In particular, we have implemented EGEON on top of OpenStack Swift [1], a highly-scalable object store.

OpenStack Swift is split into several components. The main components are the object and proxy servers. While the object servers are responsible for the storage and management of the objects, the proxy servers expose the RESTful Swift API (e.g., GET /v1/account/container/object to get object content) and stream objects to and from the clients upon request.

To be as non-intrusive as possible, the only modification we perform in the default Swift architecture is the deployment of a custom Swift middleware to intercept GET, or read, requests at the proxy servers [27]. This middleware also provides a simple API to manage the life-cycle of privacy policies. Essentially, it communicates with the Metadata Service to store and retrieve the privacy policies for protected data objects. Recall that the Metadata Service leverages Redis [15] to yield sub-millisecond access latency to metadata. To optimize request matching even further, we have collocated the Redis instance with the proxy server.

As Zion, EGEON uses containers to sandbox the execution of the chain of privacy transformations. However, contrary to Zion, which is a general-purpose serverless platform, EGEON employs a single, optimized serverless function to produce the privacy-compliant views of the underlying data. This function deserializes the compiled privacy policy, loads it into memory, evaluates the conditions from the clauses, and if it "applies", it runs the the pipeline of UDF transformations. This design has two main benefits. On the one hand, EGEON runtime starts up faster. On the other hand, UDF transformations enjoy of a great level of isolation. Simply put, they have neither direct network access nor access to the local Linux file system, among other namespaced resources, which protects the whole system from malicious transformations.

Also, to enhance response time for policies that are accessed frequently, EGEON runtime deploys a cache for policies using the Google Guava caching library [28] configured with a least-recently-used (LRU) eviction policy.

Resource allocation is managed by Zion. EGEON does not contribute any optimization at this level. When a read request comes along, our Swift middleware contacts the Zion service, which manages the containers in the object servers, and starts up a new one if necessary.

IV. EVALUATION

In this section, we evaluate key aspects of EGEON's design and our prototype implementation. First, we begin with a series of microbenchmarks to judge aspects such as system overhead, throughput and the performance of the cryptographic operators in isolation. Finally, we assess EGEON's flexibility to compose complex data views.

System setup. All the experiments have been conducted in a cluster of 8 machines: 2 Dell PowerEdge R320 machines with 12GB RAM, which operate as Swift Proxy servers, and 6 Dell PowerEdge R320 machines with 8GB RAM and 4-core CPUs, which act as object servers. The version of OpenStack Swift is Stein 5.2.0. All the machines run Ubuntu Server 20.04LTS and are interconnected through 1GbE links. The client machine for the experiments is equipped with a Intel Core i5-4440 CPU with 4 cores and 8GB RAM.

Competing systems. In some tests, we have compared EGEON against plain Zion [26] and Vertigo [29], all deployed in the same Swift cluster as above. Concretely, we have used Zion as a baseline to assess the overhead added by EGEON's software-defined architecture to the original Zion design, while we have chosen Vertigo as an example of a general-purpose, software-defined object storage system. As EGEON, Vertigo allows users to create pipelines of storage functions, each implemented as an OpenStack Storlet [30]. Hence, it is a good representative to act as a proving ground for the performance of EGEON against

a similar software-defined architecture. It must be noticed that the control plane in Vertigo is programmatic, while in EGEON is declarative through the use of JSON-based privacy policies.

A. Microbenchmarks

We have run several microbenchmarks:

Cryptographic operations. To better understand the sources of overhead incurred by EGEON, we examined the throughput of the individual transformation UDFs, since different privacy policies may result in various transformation mixes. For each type of cryptographic transformation, we measured the number of operations per second that the EGEON runtime can perform on a object server in the data plane, as well as the latency. The meaning of each operation depends on the specific type of the cryptographic transformation UDF. For HOM, this refers to the summation of two encrypted 32-bit integers. For PRE, it refers to the proxy re-encryption of a single encrypted 32-bit integer, while for SEARCH, it represents the search of a keyword in an encrypted document with the same keyword. The results of this experiment are given in Table I. As expected, we can observe that the latency of the cryptographic operations is in the order of a few milliseconds, which is acceptable for many use cases. Due to the added latency of the cryptographic transformations, a reactive data plane such as that available in EGEON, which is driven exclusively by the events appearing in the data streams, can be of great help to define privacy policies that minimize the number of cryptographic operations (for instance, by skipping unneeded data in the first steps of the transformation chain).

Overhead. To provide a full picture of the overheads incurred by EGEON, we measured the latency introduced by EGEON to the I/O path with respect to Zion and vanilla Swift. To measure the overhead, we utilized the Time to First Byte (TTFB), which captures how long the client needed to wait before receiving its first byte of the response payload from the Swift servers. To make measurements more precise, we colocated the client with one of the Swift proxy servers. As a client, we used pycurl to generate the Get requests for three different data file sizes. For each object size, we performed 1K requests. For Zion, each request caused the invocation of a NOOP function that simply echoes the input stream to the output (see Listing 1 in [26] for further details). For EGEON, we set up a NOOP policy which includes a single NOOP UDF in the transformation chain. This UDF does nothing:

```
1   package com.urv.egeon.function;
2
3   import com.urv.egeon.runtime.api.parser.UDF.ContextUDF;
4   import com.urv.egeon.runtime.api.parser.UDF.UDF;
5   import com.urv.egeon.runtime.api.parser.event.Event;
6
7   public class Noop extends UDF {
8       @Override
9       public Event update(Event e, ContextUDF ctx) { return e; }
10      @Override
11      public Object destroy(ContextUDF ctx) { return null; }
12  }
```

Listing 3: A no-operation (NOOP) UDF.

The results are given in Fig. 2. As seen in this figure, Zion and EGEON are on par, which demonstrates that EGEON software-defined architecture adds little overhead to Zion. With respect to vanilla Swift, both systems add around 9ms of extra latency, which can be considered very small. To wit, serverless function invocation in major cloud providers usually take between 25 to 320ms in warm state [31].

TABLE I: Performance of cryptographic primitives available in EGEON.

Transformation UDF	Throughput (ops/sec)	Latency (ms)
Homomorphic Addition (SUM)	616	1.62
Proxy Re-Encryption (PRE)	137	7.29
Keyword Search (SEARCH)	166	6.02

Since Zion does not support the pipelining of functions, we compared EGEON against Vertigo. Recall that Vertigo enables users to chain several Storlets together, where each Storlet can implement some reusable storage function such as decryption, compression, etc. We repeated the same experiment as above, but evaluating chains of increasing length. For EGEON, we set up chains of NOOP UDFs, while for Vertigo, we did the same, but for pipelines of NOOP Storlets. The results are depicted in Fig. 3. We can see that while the overhead keeps constant in EGEON, Vertigo shows a linear increase in latency. The reason for such a difference is the reactive core of EGEON, which does nothing if the transformations are not subscribed to any event. Vertigo, however, wires each Storlet with their neighbors in the chain, which takes some time, albeit each of them just copies the input to the output. This strongly reinforces the idea that for a software-defined data protection system to be useful, it is not a good idea to route the data streams through a pipeline of functions, but rather to act on them when it is strictly needed. Indeed, Vertigo's overhead is more than one magnitude higher than EGEON's overhead as shown in Fig. 3.

Throughput. As a final microbenchmark, we quantified the impact of EGEON on the system throughput. As above, EGEON was compared to Zion and vanilla Swift to give real sense of its performance. For this experiment, we utilized the getput benchmarking tool suite [32] for Swift. And in particular, the gpsuite to conduct parallel tests with multiple clients. More concretely, we run gpsuite in one of the Swift proxy servers for 10 seconds and for different object sizes. We considered a replication factor of 3, and instrumented gpsuite to stress 3 out of the 6 object servers in the data plane. As in the overhead test, a NOOP function call per request was made for Zion and a NOOP policy plus NOOP UDF for EGEON. Table II reports the maximum throughput in operations per second attained by each system. Similarly to what was observed for the overhead, EGEON and Zion perform in similar terms. More interestingly, as the object size increases, the gap between both EGEON and Zion and Swift grows. We investigated this issue and we found that this happens due to a higher CPU interference caused by the JVM used to run the EGEON logic and the NOOP code in Zion, respectively.

Also, Fig. 4a plots the throughput for an increasing number of emulated clients for a 1MB object, which exhibits the same behavior as before. That is, EGEON and Zion showing a similar performance, while Swift delivering a much higher throughput due to the absence of any computation in the I/O path. Finally, Fig. 4b illustrates EGEON's slowdown relative to vanilla Swift, calculated as Slowdown $= \frac{\text{EGEON download time}}{\text{Swift download time}}$, as a function of the object size (x-axis) and the number of concurrent clients (y-axis). As can be seen in the figure, the slowdown factor does not increase steeply. Rather, it increases gradually in both axes, never doubling the latency. This indicates that pushing down data protection logic to the storage may be acceptable for many applications.

(a) TTFB for a 10KB object.　(b) TTFB for a 100KB object.　(c) TTFB for a 1MB object.

Fig. 2: Time to First Byte (TTFB) for different object sizes.

(a) Overhead of EGEON.　(b) Overhead of Vertigo.

Fig. 3: Overhead of chain setup of EGEON versus Vertigo.

TABLE II: Maximum throughput (ops/sec) for different object sizes.

System	Object size		
	100KB	1MB	10MB
Swift	157,81	111,43	31,62
Zion	140,72	79,69	21,17
Egeon	140,56	79,59	21,26

B. Applications

To evaluate the composability of EGEON, we have designed two privacy policies that capture the different complexities of real-world applications. These applications are the following:

Covid-19 use case. In this use case, we demonstrate the same policy of Listing 1, but applied to healthcare. We use the JSON file reported by the US government that summarizes the patient impact on healthcare facilities caused by Covid-19 [33] (April 2021). Specifically, we exchange the user label "treasurer" by "state coordinator", and label the field that reports the sum of patients hospitalized in a pediatric inpatient bed in 7-day periods [FAQ-10.b)] as "sensitive", as it reveals which hospitals may be collapsed. The rest of information is ignored. As a result, the transformed view for state officials only bears a single homomorphically encrypted value that aggregates the sum of all healthcare facilities. As in Listing 1, the policy links 3 UDFs: CLAC→HOM→PRE. To play out with the file size, we split this dataset into three smaller files based on increasingly smaller time periods: year, month and week.

Adult dataset [34] use case. This dataset in CSV format from the UCI Machine Learning Repository [34] has 48842 records and 14 attributes. Some of these attributes can leak sensitive information such as race and occupation. For this use case, we have decided to protect the attribute #7: occupation, with the SEARCH scheme to allow for type-of-employment searches on

encrypted text. To prove composability, we have used CLAC to protect three attributes out of the four attributes chosen for this experiment. We have used one object label: "sensitive", and one user label: "HR manager", so that only a human resources manager can retrieve the three protected columns. Further, we have encrypted other fields to increase the file size to 134MB, to later split it into 3 smaller files based on the attribute #6: marital-status. The policy is as follows (some fields have been omitted for brevity):

```
{
"Object": "v1/{account}/{container}/adult.csv",
"Action": {
 "StartAt": "Step1",
 "Steps": {
  "Step1": {
   "Id": "CLAC",
   "EventType": {
     "Type":   "ColumnMarkerEvent",
     "Input": [{"columns": [2,6,7], "olabel": "sensitive" }]
   },
   "Input": [{"ulabel": "HR manager", "olabel": "sensitive" }],
   "Next": "Step2"
  },
  "Step2": {
   "Id": "SEARCH",
   "EventType": {"Type": "ColumnEvent", "Input": [{"column":7}]},
   "Next": "End"
}}}
```

TABLE III: Network Speeds.

Network	4G	Fiber	LAN
Median speed (Mbps)	28.9 [35]	55.98 [36]	887 (SpeedTest)

Experiment. In this test, we measure the time to download the raw files directly from Swift against the time to download the transformed views generated by EGEON. The goal is to decide if it is worth to push the privacy transformations into the object store instead of running them on the user devices and VMs, so that software-defined data protection is within reach. To do so, we have capped the ingoing bandwidth of our client machine to emulate different network speeds and customary scenarios: Fiber network speeds to emulate home and business users, 4G network bandwidth to emulate mobile users, and finally, LAN to simulate a scenario where the client and Swift servers reside on the same local area network (e.g., a university intranet). The exact network speeds are listed in Table III. We performed 1K executions per object size and network speed.

The results are plotted in Fig. 5a for the Covid-19 use case, and in Fig. 5b for the Adult dataset. Error bars display the standard deviations of results, which are indeed very narrow. Non-surprisingly, we can see that EGEON lowers the download time significantly for the slow 4G and fiber connections, which

(a) Throughput (ops/sec) for a 1MB object.

(b) Slowdown of EGEON over vanilla Swift.

Fig. 4: Throughput and latency of EGEON under a multi-client setting.

(a) Covid-19 use case.

(b) Adult dataset use case.

Fig. 5: Performance of EGEON in two composite policies.

means that pushing down privacy transformations to storage is a good deal better than the naive approach of encrypting data on the client side and retrieve the whole file as alleged by cloud providers such as AWS for S3. The savings in some scenarios can be dramatic such as in the Covid-19 use case, where just a few bytes (e.g., aggregates such as SUM, COUNT and AVG, etc.) are consumed by the application, reaching 72.1X speedup for 4G mobile terminals. For the LAN setting, the benefits are not so clear, and for the Adult dataset, EGEON shows a slowdown factor of 3.3X in the worst case due to the heavy computations associated with keyword search—actually, the test primitive of SEARCH requires three pairing operations [24]. Either way, we believe that EGEON's fine-grained data protection capabilities outweigh the slight loss of performance.

V. RELATED WORK

Software-defined storage systems. A first category of related work comprises software-defined approaches for storage, and in particular, for object storage systems. The common feature of these approaches is that they break the vertical alignment of conventional storage infrastructures by reorganizing the I/O stack to decouple the control and data flows into two planes of functionality—control and data. A number of proposals have followed this approach, including IOFlow [37], sRoutes [38], Retro [39], Vertigo [29] and Crystal [13], [14].

Among them, only Vertigo and Crystal have been tailored to object storage. As EGEON, both systems have been deployed atop OpenStack Swift. But unlike EGEON, their data plane is based on OpenStack Storlets [30]. A Storlet is a piece of Java logic that is injected into the data plane to run custom storage services over incoming I/O requests. As in EGEON, this design increases the modularity and programmability of the data plane stages, fostering reutilization. However, Storlet-made pipelines

are not reactive, wasting resources when we are only interested in specific elements of the data stream. Moreover, the Storlet-enabled data plane of Vertigo and Crystal emphasizes control-flow over data-flow, making it hard to explicitly represent the (cryptograhic) transformations of objects. Per contra, EGEON's data plane is driven solely by the events showing up in the data streams, which makes it easy to reuse the same transformations over and over again on different data. Only the events must be re-defined in the policies.

Finally, it is worth to note that to the best of our knowledge, we are not aware of another software-defined storage system that automatically enforces privacy policies along the I/O path as EGEON. Software-defined security has remained within the boundaries of software-defined networking (see, for instance, Fresco [40]). The only exception is the recent vision paper [41] on software-defined data protection. Like EGEON, [41] argues that the key ideas of software-defined storage can be translated to the data protection domain. However, the approach of [41] is radically different. Instead of adding privacy controls to the I/O stack of a disk-based object storage system, [41] assumes all in-storage processing to occur on FPGAs and "smart storage" devices, for we see [41] as an orthogonal work to us.

Privacy Policy Enforcement. There exist many systems that enforce privacy policies automatically. Most of these systems resort to Information Flow Control (IFC) as a means to control how information flows through the system. See, for instance, Riverbed [42], which uses IFC to enforce user policies on how a web service should release sensitive user data. In contrast to these systems, EGEON follows a software-defined approach to leverage the storage resources and enforce the privacy policies where data is. Similar to our transformation chains, Zeph [43] proposes to enforce privacy controls cryptographically but over encrypted stream processing pipelines. Specifically for storage, Guardat [44], at the block level, and Pesos [45], at the object level, enable users to specify security policies, for instance, to stipulate that accesses to a file require a record be added to an append-only log file. Nonetheless, these systems do not permit the composition of advanced privacy controls as EGEON, and thus, fall short to empower users with strong data controllers.

VI. CONCLUSIONS

As increasingly much more sensitive data is being collected to gain valuable insights, the need to natively integrate privacy controls into the storage systems is growing in importance. In particular, the poor interface of object storage systems, which

lacks of sophisticated data protection mechanisms, along with the inherent difficulties to refactor them, have motivated us to design and implement EGEON. To put in a nutshell, EGEON is a novel software-defined data protection framework for object storage. It allows data owners to define privacy policies on how their data can be shared, which permit the composition of data transformations to build sophisticated data protection controls. In this way, data owners can specify multiple views from the same data piece, and modify these views by only updating the policies (e.g., by modifying the chain of transformations that produce a particular view), leaving the system internals intact. The EGEON prototype has been coded atop OpenStack Swift. And our evaluation results demonstrate that EGEON adds little overhead to the system, yet empowering users with the needed controls to ensure strong data protection.

REFERENCES

[1] OpenStack, "Swift," https://docs.openstack.org/swift/, 2021.
[2] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in *19th ACM/IFIP Middleware Conference Industry (Middleware'18)*, 2018, pp. 1–7.
[3] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: scalable analytics on serverless infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, 2019, pp. 193–206.
[4] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*, 2020, pp. 115–130.
[5] M. Sánchez-Artigas, G. T. Eizaguirre, G. Vernik, L. Stuart, and P. García-López, "Primula: A practical shuffle/sort operator for serverless computing," in *21st International Middleware Conference Industrial Track (Middleware'20)*, 2020, pp. 31–37.
[6] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 285–301.
[7] Amazon, "Amazon S3 Developer Guide," http://aws.amazon.com/documentation/s3, 2018.
[8] IBM, "IBM Clod Object Storage," https://cloud.ibm.com/docs/services/cloud-object-storage, 2018.
[9] P. Biswas, F. Patwa, and R. Sandhu, "Content level access control for openstack swift storage," in *ACM CODASPY*, 2015, pp. 123–126.
[10] X. Li, R. Li, P. P. C. Lee, and Y. Hu, "OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems," in *USENIX FAST*, 2019, pp. 331–344.
[11] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *USENIX OSDI*, 2006, pp. 307–320.
[12] L. Militano, "Experimenting on Ceph Object Classes for Active Storage," https://blog.zhaw.ch/icclab/experimenting-with-ceph-object-classes-for-active-storage/, 2019.
[13] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom, "Crystal: Software-defined storage for multi-tenant object stores," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 243–256.
[14] R. Gracia-Tinedo, J. Sampé, G. París, M. Sánchez-Artigas, P. García-López, and Y. Moatti, "Software-defined object storage in multi-tenant environments," *Future Generation Computer Systems*, vol. 99, pp. 54–72, 2019.
[15] "Redis," https://redis.io/.
[16] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
[17] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, aug 2013.
[18] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011, pp. 85–100.
[19] W3C, "XML Path Language," https://www.w3.org/TR/xpath-30/, 2020.
[20] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy, "Secure sharing of partially homomorphic encrypted iot data," in *15th ACM Conference on Embedded Network Sensor Systems (SenSys'17)*, 2017.
[21] F. Vercauteren, "Optimal pairings," *IEEE Transactions on Information Theory*, vol. 56, no. 1, pp. 455–461, 2010.
[22] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *Selected Areas in Cryptography (SAC)*, 2006, pp. 319–331.
[23] D. F. Aranha and C. P. Gouvêa, "Relic is an efficient library for cryptography," https://github.com/relic-toolkit/relic, 2017.
[24] Y. H. Hwang and P. J. Lee, "Public key encryption with conjunctive keyword search and its extension to a multi-user system," in *International conference on pairing-based cryptography*. Springer, 2007, pp. 2–22.
[25] A. De Caro and V. Iovino, "jpbc: Java pairing based cryptography," in *16th IEEE Symposium on Computers and Communications, ISCC 2011*, 2011, pp. 850–855.
[26] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in *18th ACM/IFIP/USENIX Middleware Conference (Middleware'17)*, 2017, pp. 121–133.
[27] OpenStack, "Middleware and metadata," https://docs.openstack.org/swift/latest/development_middleware.html, 2021.
[28] Google, "Guava: Google Core Libraries for Java," https://github.com/google/guava, 2021.
[29] J. Sampé, P. García-López, and M. Sánchez-Artigas, "Vertigo: Programmable micro-controllers for software-defined object storage," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD'16)*, 2016, pp. 180–187.
[30] OpenStack, "Welcome to storlets' documentation!" https://docs.openstack.org/storlets/latest/, 2020.
[31] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 133–146.
[32] M. Seeger, "GetPut benchmarking suite," https://github.com/markseger/getput, 2021.
[33] U.S. Department of Health & Human Services, "COVID-19 Reported Patient Impact and Hospital Capacity by Facility," https://healthdata.gov/Hospital/COVID-19-Reported-Patient-Impact-and-Hospital-Capa/anag-cw7u, Apr. 2021.
[34] C. B. D. Newman and C. Merz, "UCI repository of machine learning databases," https://archive.ics.uci.edu/ml/index.php, 1998.
[35] www.opensignal.com, "Benchmarking the global 5G user experience – October update — Opensignal," https://www.opensignal.com/2020/10/13/benchmarking-the-global-5g-user-experience-october-update.
[36] www.fcc.gov, "Measuring Fixed Broadband - Tenth Report — Federal Communications Commission," https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-tenth-report.
[37] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: a software-defined storage architecture," in *ACM SOSP'13*, 2013, pp. 182–196.
[38] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska, "sRoute: treating the storage stack like a network," in *USENIX FAST'16*, 2016, pp. 197–212.
[39] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *USENIX NSDI'15*, 2015, pp. 589–603.
[40] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks." in *20th Annual Network and Distributed System Security Symposium NDSS'13*, 2013.
[41] Z. István, S. Ponnapalli, and V. Chidambaram, "Software-defined data protection: Low overhead policy compliance at the storage layer is within reach!" *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1167–1174, 2021.
[42] F. Wang, R. Ko, and J. Mickens, "Riverbed: Enforcing user-defined privacy constraints in distributed web services," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 615–630.
[43] L. Burkhalter, N. Küchler, A. Viand, H. Shafagh, and A. Hithnawi, "Zeph: Cryptographic enforcement of end-to-end data privacy," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 387–404.
[44] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post, "Guardat: Enforcing data policies

at the storage layer," in *Tenth European Conference on Computer Systems (EuroSys '15)*, 2015.

[45] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "Pesos: Policy enhanced secure object store," in *Thirteenth EuroSys Conference (EuroSys '18)*, 2018.