Implementing Data Flow Assertions in gRPC and Protobufs – Final Report

Akshat Mahajan
Brown University

Yingjie Xue
Brown University

Jonathan Weisskoff
Brown University

Abstract

Information flow control ensures programs handle sensitive data securely across service boundaries. In this paper, we implement information flow control for microservices-oriented architecture using modified versions of the popular remote procedure call framework known as gRPC. In our implementation, messages sent via the networks are first encrypted when they arrive at any microservice. We enforce policy checking at two places: (1) when functions within a microservice try to access or modify the message, and (2) when a microservice handles requests from clients and other microservices. If the policy checking is not passed, access to plaintext values is disallowed. We benchmark its impact on performance with a microservices-oriented application built by Google, and provide examples of use with simple demonstrative applications.

1 Introduction

The General Data Protection Regulation (GDPR)'s requirements for data processors, as instituted by the European Union, give companies incentive to enforce rules against improper data processing. Preventing improper data processing is complicated by the distributed nature of data processing solutions today. We are motivated by two situations where user data may be improperly used in modern architectures:

- **Legacy code paths operating against user preferences**. A user may want some of their data to be unavailable to some system components — for example, they may prefer their IP address not be used for ad services or their website activity be withheld from machine learning training models. Honouring these commitments may impose nontrivial work on system maintainers in large codebases.
- **Accidental leaks to other services**. Sensitive data may leak to log services or monitoring solutions owing to unforeseen code paths (such as an error message which prints out personal identifying information and is then captured by an error aggregation service). They can also leak through improper safeguards — for example, a misconfiguration might cause applications processing European data to communicate with US databases rather than EU databases.

The likelihood of both issues are exacerbated by the microservices pattern, where new core functionality is outsourced to a new service rather than incorporated into an existing one. Each new service adds potential failure modes for the entire system, and raises issues about protecting data across a complex network of service dependencies.

Verifying at runtime whether access to sensitive data is permitted can solve both of these issues by preventing unwanted access to data. We propose to automatically generate and embed such *data flow assertions* into gRPC, a remote procedure call framework built on top of the serialization framework known as protocol buffers [2]. Data flow assertions allow organizations to secure data even as their architecture evolves or degrades with minimal changes to their codebase. Our main contributions are as follows:

- We wrote and embedded a Go library (hereafter referered to as a *privacy* library) that allows gRPC, protocol buffers and other packages to perform runtime assertions about messages within and between service boundaries. Its API serves as a reference implementation for similar ports to other languages.
- We modified gRPC and protocol buffers (only for Go) in several places to ensure all messages generated and transmitted by these frameworks is protected by our privacy library. Introducing privacy is thus as simple as using our modified gRPC and protocol buffer binaries instead.
- We benchmark our modifications using a sample microservices-oriented demo application [3] built by Google Cloud Platform, and provide commentary on the performance of our design. We suggest performance improvements for future iterations.

2 Related Work

Projects like Resin [5] and Jacqueline [4] augment server backends to prevent unauthorized access to data from end users or system components. Our project similarly aims to protect leaks of personally identifiable information (PII) in server backends, but with two key differences:

1. We recognize that modern web architecture is built around *microservices*: independent lightweight web services that communicate to produce a final state for the end user of a website. Microservices eliminate guarantees about technology stacks in a request's path, complicating traditional information control substantially — it is no longer sufficient to augment one particular framework or language runtime to achieve privacy guarantees. We believe we are the first to address this problem meaningfully by altering the API boundary instead of the compiler directly.

2. We operate on a comparatively relaxed threat model. We focus purely on *accidental* leaks i.e. where no actors are necessarily malicious, but where the leak nevertheless violates organizational rules or the law. Common examples of such leaks includes displaying PII in server logs on encountering an error, communicating with an insecure service owing to misconfiguration, accidentally storing data about a country's citizens outside the resident country (violating data residency laws such as Australia's), etc. These carry penalties for most organizations and are detrimental for customers' safety.

Our scheme takes advantage of gRPC [1], which is a remote procedure call framework built on top of protocol buffers [2], a serialization framework. Protocol buffers generate stub code in multiple languages for message serialization/deserialization — gRPC then uses this stub code to communicate with other gRPC servers across the network, using either *unary* handlers (a one-time connection opened for receiving and sending a single message) or *streaming* handlers (a bidirectional channel between client and server for communicating multiple messages over an active communication). This gives gRPC a unique bird's eye view of all data entering and leaving the service, as well as the ability to validate if message targets will honour the appropriate requirements for handling data. gRPC and protocol buffers themselves exist as API specifications — they are implemented separately for each language they support. Any changes to the core specification must be propagated to each of these implementations individually. Given the complexity of our feature, changing gRPC for every language is nontrivial — after tinkering with several different gRPC language implementations, we chose to focus on Go specifically owing to its prominence in our chosen benchmark application. Consequently, all microservices that employ our contributions must be written in Go — however, our design does not assume anything specific to Go in its core concepts, and we believe porting to other languages is relatively straightforward.

3 Design and Implementation

Our chief design goal was to ensure backwards-compatibility — adopters of our techniques should not have to rewrite their code significantly to reap the benefits of our tooling. Fig. 1 describes our system architecture in brief. We accomplish our runtime assertion scheme by:

1. automatically *encrypting* personally identifiable information when gRPC receives a message. For our prototype implementation, we encrypt all strings within all messages by default — we discuss the impact of this design choice in our evaluation section.

2. augmenting all protocol buffer messages with specialized getter and setter methods. These getter and setter methods are automatically inserted when a **.proto** file is converted into Go code, and embed wrappers that check if the appropriate get or set is permitted in a given context.

3. introducing default server-side interceptors into gRPC. Interceptors are middleware that receive messages prior to passing it on to server code. Our interceptors inspect metadata (e.g. HTTP headers) sent by a request to decide whether or not these requests are appropriate.

4. defining a JSON policy file that is read once at microservice launch. This policy file specifies the conditions under which our augmented getter and setter method should successfully complete, as well as the conditions our server-side interceptors are meant to validate.

3.1 Encryption Mechanics

Encryption in-memory forms the bedrock of our approach to security here. It doubles as a form of *masking* (desensitizing data when being shared) as well as a straightforward defense against bypassing our library's protections — by storing only the encrypted form and disabling raw access to the key, calling code is forced to use our privacy library API to retrieve the plaintext value.

We make use of the Fernet cryptography suite for symmetric encryption, generating a private key exactly once in memory (on library import) and encrypting all message fields with this key. This design means every microservice generates its own private key — messages encrypted in one microservice cannot be decrypted within another microservice. To prevent breaking upstream services, we decrypt messages before transmitting them, relying on the organization to ensure secure transfer of messages, and then re-encrypt once they reach their destination microservice. It is possible to request *not* decrypting a message field before converting to wire-format and transmitting using our policy file.

Currently, our key handling mechanisms are limited — in future iterations, we would like to support loading keys externally and allowing manual rotation.

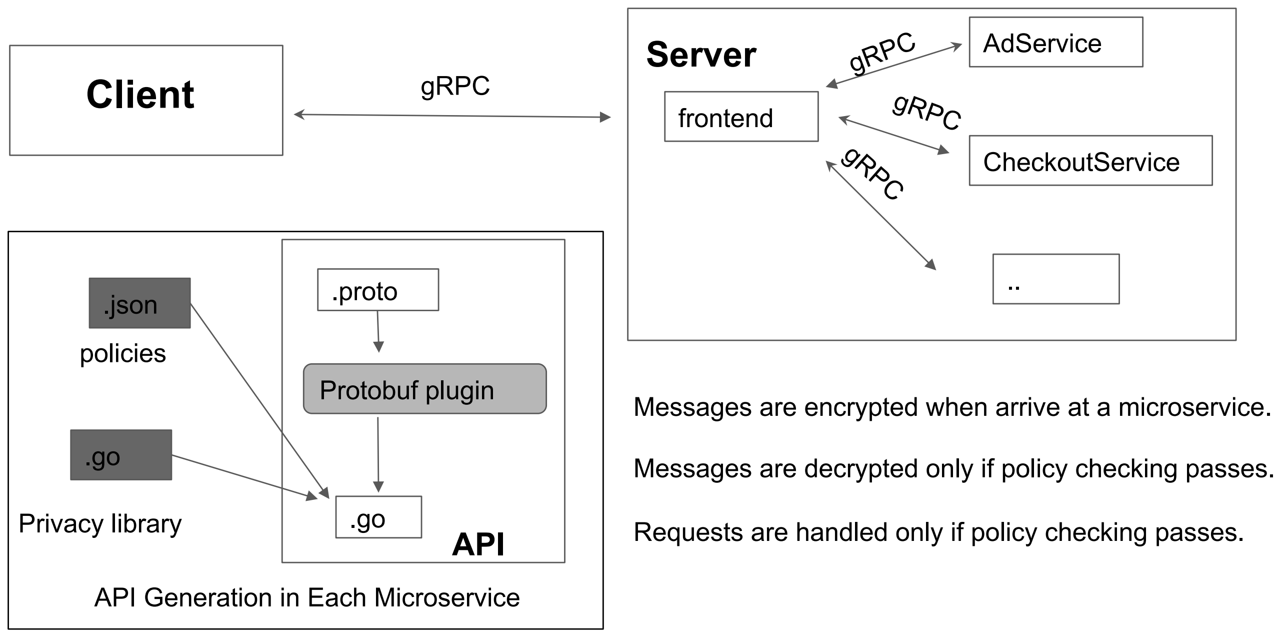

Client
gRPC
Server
gRPC
AdService
frontend
gRPC
CheckoutService
gRPC
..
.json
policies
.proto
Protobuf plugin
.go
Privacy library
.go
API
API Generation in Each Microservice
Messages are encrypted when arrive at a microservice.
Messages are decrypted only if policy checking passes.
Requests are handled only if policy checking passes.

Figure 1: System Architecture. The *API* box describes an offline build-time stage where RPC specifications are converted into pluggable Go code, which is where we inject our privacy library as well as our policy files for reference. The *server* box represents an *online* runtime stage within our sample benchmark application: a single FRONTEND service contacted by the client (e.g. a browser) communicates using gRPC to a variety of services, utilizing our injected library on each invocation.

3.2 Enforcing Privacy Checks within A Microservice

Our library models any operation against an encrypted field as a type of predefined *verb* or *action*. Every action can then have conditions statically specified against it by the developer — if the policy passes, the action is allowed, otherwise the action returns the ciphertext stored in memory for the field. Table 1 documents the COPYING, PRINTING, and MODIFYING verbs in our policies for Go code, and the actions they are supposed to represent whitelisting — more verbs can be added when porting to other languages.

All getter and setter methods for field members in our implementation pass along information about the kind of action being performed to our library for appropriate checks — for example, a call to GETFOO() may look up permissions under the COPYING verb of our privacy policy, and both serializing data to wire-format and to string representation look up conditions for PRINTING. These verb lookups are hardcoded in generated protocol buffer code.

Currently, our policy verbs are not as granular as we would like — addition, modification, or subtraction in Go must be modelled by defaults like COPYING rather than dedicated ADDING, MULTIPLYING, or SUBTRACTING methods. The chief blocker towards implementing dedicated verbs is absence of operator overloading support — in porting to languages with operator overloading support, we envision richer and more expressive policies will be possible.

Table 1: Verbs in our policy specification and their meaning

Policy Verb	**Operations It Models**
`Copying`	Default for operations that require access to the underlying value but aren't specifically whitelisted by other actions. e.g. string concatenation, reads, splits, etc.
`Modifying`	Any operation that may potentially modify the underlying value. e.g. pointer access (currently unchecked in our implementation), explicit sets, etc.
`Printing`	Whether or not to display a field when the entire message is type-cast to a different representation. Includes both string conversion or protobuf-wire conversion.

Listing 1: Policy example (see section 3.2)

```
1  {"Policies": [
2      {
3        "Message": "HelloRequest",
4        "Field": "Name",
5        "Conditions": {
6          // GetName() will only succeed
7          // in trustedFunc in main
8          "Copying": [
9            {
10             "allowed": true,
11             "if": "main.
                    trustedFunc < main.main"
12           }
13         ],
14         // modifying is never permitted
15         "Modifying": []
16         // printing is always permitted
17         // because it is absent in file
18       }
19     }],
20   // explained in section 3.3
21   "RequestValidation": ...
22 }
```

Once informed an operation is being performed against data, our policy file allows developers to specify the *context* that operation is allowed to succeed in. Our most innovative contribution here is the notion of *function order* as a policy context. A developer can simply specify a partial ordering of fully-qualified function names as a condition for an action — this partial ordering is compared to the call stack at runtime, and if the relative order of these functions is reflected in the call stack, the operation can be failed or succeeded per the developer's wishes.

Specifying a string `foo.untrustedFunc < foo.main`, for example, indicates that all invocations of an operation inside `untrustedFunc` should succeed only if `untrustedFunc` was invoked in the `main` function of the Go package `foo` and nowhere else. This is a simple way to define trusted code paths — it works across multi-threaded code, greenlets, re-entrant functions, and other exotic paths because the call stack is never corrupted. This is much simpler than Resin's data tracking, achieving most of its goals without the need to know where data has been.

There are important limitations to this approach compared to Resin-style data tracking that are worth mentioning. First, relying on function name means we cannot support anonymous functions. Second, context information not on the current call stack is inexpressible — disallowing access specifically if a branch path was taken in the function somewhere, for example, is not possible. Third, specifying overlapping chains (allowing if inside *main*, but disallowing if inside *bar* in *main*) is possible with this approach, meaning there is potential for conflict. Writing and using named helper functions where appropriate resolves the first two of these limitations. Our policy syntax further solves potential conflicts by allowing developers to express precedence between different conditions — whichever condition is first in the conditions array (see lines 10 in Listing 1) gets higher precedence.

Listing 1 provides a concrete reification of these concepts in our policy file.

3.3 Enforcing Privacy Checks Between Microservices

gRPC allows developers to define client-side and server-side middleware functions known as *interceptors*. These functions can read, modify and perform authorization checks on incoming messages before any server-side code is allowed to handle it. As such, they are well-placed to make flow assertions before data is handled by any code.

Interceptors are an optional feature, often written directly by developers. For our prototype implementation, we modified gRPC to have default interceptors that scan sent metadata about a request and compare it to declarations in our policy file, if any. This metadata is inserted on the client side by developers directly using gRPC's inbuilt `metadata` package, giving them the flexibility to develop complicated per-request metadata logic. This model forms a robust foundation for more complex and generic permissioning schemes that can take the domain model for any application into account, such as one that maps user preferences or relationship to data to whether or not a request should be transmitted to a specific service — as we cannot predict the domain model for any arbitrary usage of this library in advance, we refrain from introducing any specific permission schemes ourselves. An example of how our interceptor in action can prevent some misconfiguration issues is described in our evaluation section.

Currently, our interceptors can only statically validate exact matches of metadata against our policy file — in other words, we can only check if a specific value of a specific key exactly matches the key and value specified in our policy file. We anticipate there is value in allowing for much more flexible examinations of metadata in future iterations.

Code snippet 2 showcases how we can express the desire to exactly match key-value pairs in metadata for requests to an endpoint concisely.

4 Evaluation

The core implementation for data flow assertions above clocks in at just 660 lines contained in a single repository here. We contributed another ≈ 500 lines of additional changes to gRPC (here and here) and protocol-buffers here to implement automatic generation of getter and setter methods.

Listing 2: Policy file snippet (see section 3.3 for explanation of terms)

```
{"RequestValidation": [{
    // server endpoint name
    "MethodName":
        "/helloworld.Greeter/SayHello",
    // what to check in metadata
    "MatchingContext": {
      "region": ["us-east-1"]
    }
  }],
  // explained in section 3.2
  "Policies": ...
}
```

To verify the viability of our approach, we broke evaluation into two phases: functionality validation with a toy `helloworld` example here, and load benchmarks with a more complicated suite of microservices here.

4.1 Functionality

We copied a `helloworld` example provided directly by Google. This example features just two services — a client and a server. Our goal was to verify whether the following policies could be expressed:

1. Disallowing an unsafe logger method from gathering access to data within an otherwise safe function.
2. Disallowing messages sent from another regional datacenter by accident

We began by regenerating the Go code produced by protocol buffers for this example using our modified gRPC and protocol buffers. These generated files gained ≈ 100 more lines of code, but all of this code was automatically produced by one command.

To implement the first case, we first wrote an "unsafe" method that attempted to simply convert a received message into log lines, and inserted it into the main server handler code within the server. We then wrote a simple privacy policy here that explicitly disallows all attempts to cast a message to a string representation within this unsafe logger function. After loading it with an environment variable `GRPC_PRIVACY_POLICY_LOCATION`, we were able to see that log lines for just this message in this unsafe logger published the masked (encrypted) value of the message, while other operations on the data were not hindered.

To implement the second case, we added `RequestValidation` statements from Listing 2 to our policy file. This made the server pretend it only accepted requests that indicated they originated from region `us-east-1`. We then modified the client to send metadata indicating they originated from a different region (say, `us-east-2`). We were able to observe the client being rebuffed (the connection was closed by the server without sending any data) and that the server logic never processed the received request.

Aside from the introduction of our logger function and our one-step regeneration of the protocol buffer, our changes to accomplish this were absolutely minor.

4.2 Load Performance

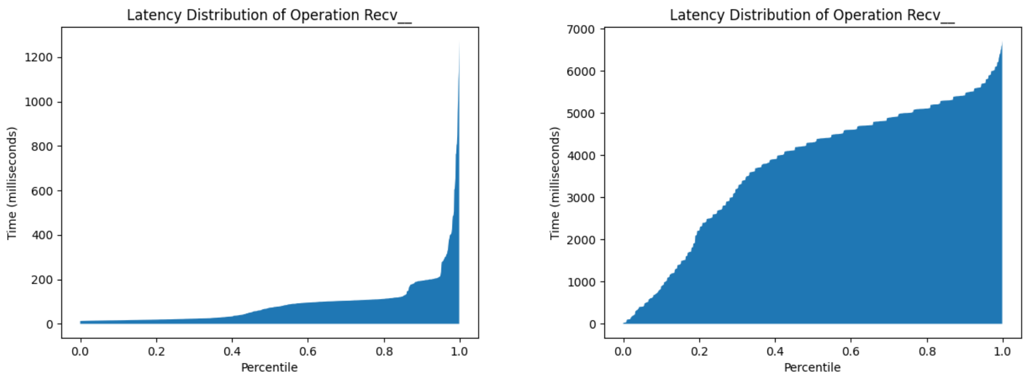
We attempted to verify just how much effort would be needed to implement policies for a significantly larger demo application provided by Google here. Clocking in at approximately 70,000 lines of code split across ten services, this sample demo implements a shopping application for an online e-commerce store and uses at least four Golang-based microservices, most notably for rendering the website itself for client requests. It also comes with inbuilt instrumentation for distributed tracing with Jaegar and full tooling for deploying to a remote Kubernetes cluster, simplifying our time to test considerably.

We observed that naively switching from standard gRPC to our newer version for this application was not painless — certain core functionality, such as routing and page templating, were affected in the migration. This was expected, as much of the underlying codebase accessed message field values directly and would break since our implementation now returned encrypted values on these accesses. To address this, we refactored such direct access to use our provided getter and setter methods instead, resulting in approximately 50 lines of code being modified across four services.

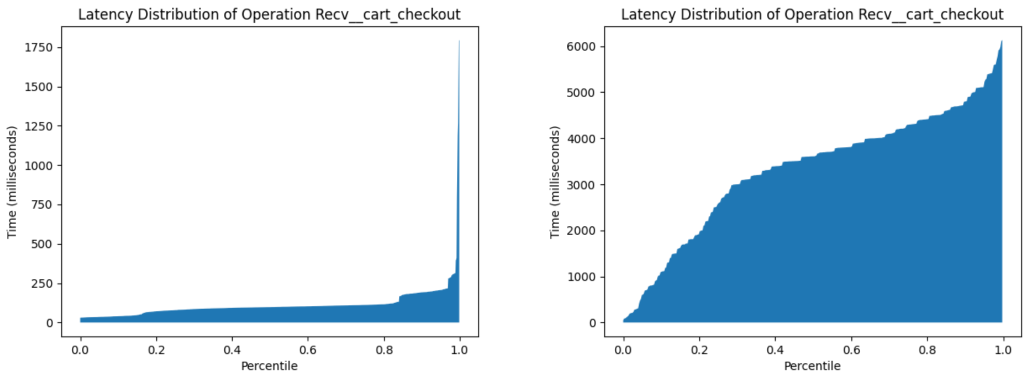
One area where refactoring could not address the underlying cause was in HTML templating — the templating engine would embed lookup directives inside the HTML template and naively access the fields of the variables it was provided according to these lookup directive. Since we did not have access to the underlying template engine's code, we instead had to import our privacy library directly in the application and recursively decrypt the message passed to the template engine — this itself was just a two-line change.

We verified using a local Kubernetes cluster that our application worked successfully following these two sets of changes. To measure load performance, we did the following prerequisites:

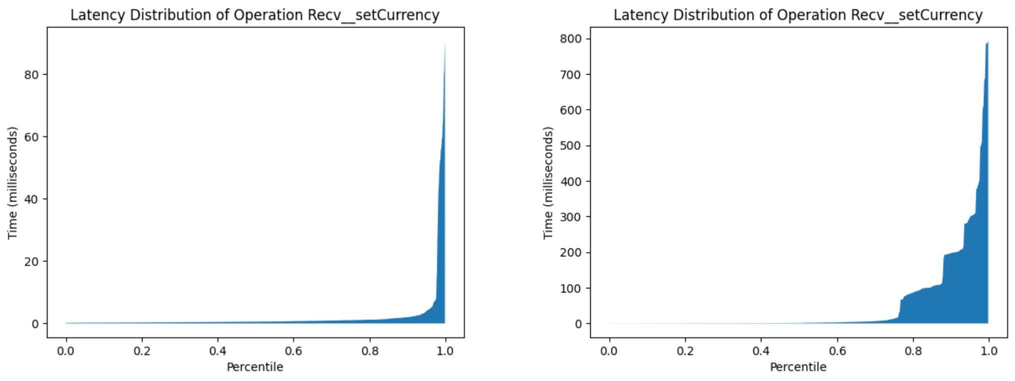
1. Launched a Kubernetes cluster on Google Kubernetes Engine consisting of three n1-machine-2 instance types. These instance types are relatively underpowered compared to high-end quadcore laptops, possessing just 2 vCPUs and 7.5 GBs of memory. We chose this instance type as reflective of real-world clusters serving traffic per the authors' own experiences.
2. Installed Jaegar on our operating cluster, and configured a load generating service to saturate six endpoints of our application with equal frequency at the rate of 100



(a) Latency distribution for requests to the home page (triggers queries to all services)



(b) Latency distribution for requests, to the Golang cart service, from the Golang frontend server



(c) Latency distribution for requests, to the Javascript-based currency service, from the Golang frontend server

Figure 2: (*left*) Baseline results (*right*) Benchmark results

concurrent requests per second. We waited until at least one endpoint had collected 1000 unique traces. Our endpoint choice targeted two product pages, the home page, GET requests for the shopping cart page (which was not backed by a Golang service and thus not modified, POST request fetches to checkout the cart, and a GET request to set the appropriate currency (which was not a Golang service and thus was not modified for our benchmarks), giving us a good characterization of performance impact between two Golang services, one Golang service and a non-Golang service, and between different request types.

3. Programmatically scraped trace results from Jaegar's query interface and constructed latency percentile graphs for each endpoint by tabulating the total breakdown.

We ran this setup twice, once with a baseline version (no modifications made to any service, all operating with standard gRPC) and our test version (four services modified, all using modified gRPC). This test version ran with no policies in place — we planned to add benchmarking for policies, but observed the performance impact was sizable enough that we would not be able to characterize the policy impact appropriately.

We observed a 10 - 100x slowdown across nearly all endpoints between our baseline and test versions, including endpoints such as `setCurrency` for services that had not been modified. The largest response time seen across all endpoints clocked in at about 10 seconds compared to a maximum of 1.2 for our baseline versions — conversely, we observed that the lowest response time remained close to zero for nearly all requests. Figure 2 presents the latency numbers for three of our instrumented operations, chosen for their representative nature of our results overall.

We believe that this pattern points to either CPU contention or I/O contention for concurrent encryption and decryption along a request's hot path — at small user rates, at or around 1 request per second, no observable latency discrepancy is easily noticed between baseline and target clusters. We attempted to validate this hypothesis by testing with a slightly larger cluster — 5 n1-machine-4 instances, featuring 15 GBs and 4 vCPUs each — and found minor reductions in median latency compared to our target cluster that are nevertheless in the quoted 10 - 100x range compared to our baseline cluster.

4.3 Points of Improvement

We argue that our core design could benefit from two technical improvements we were unable to implement in our prototype stage:

1. Not requiring mandatory encryption of *all* messages on receipt. This design choice was motivated by the need to prevent bypassing our user library through either inference attacks or raw memory access as well as a technical limitation in gRPC that prevents scoping encryption to just specific fields, but adds an immense amount of overhead. An alternate design could instead only encrypt specific fields on message receipt, using other mechanisms to shield unwanted access to plaintext values in memory.

2. Data tracking improvements. Our implementation currently eschews any explicit data tracking, preferring to grab the *context* for an operation event to determine whether an operation should be allowed. However, this approach, although much simpler to implement, trades off full coverage of every edge case without deeper instrumentation. We suggest a hybrid solution is to *require* distributed tracing in most applications and let privacy checks inspect the associated baggage in the surrounding trace to obtain richer information about code paths — alternatively, make use of eBPF to publish user-defined trace events when accessing core Golang structs, allowing developers to see where privacy leaks are occuring but trading off support for preventing leaks at runtime.

5 Division of Work

Most of the contribution can be seen from the Github. However, some discussions, system designs and group efforts are not shown. Akshat is the leader of the project and he has done most of the work, including writing the code for privacy library, modify protobuf-go and grpc-go and evaluate our implementation. Yingjie is responsible for the policy module design and she takes care of modifying the microservice-demo we use to make it work exactly as the original one. Regarding the policies, she contributed partial code regarding policy design and policy loading in the privacy library. Since our modified microservice-demo application is much slower than the benchmark, which is sufficient to show how much slowdown we introduce, the policies we had for the microservices demo are not used in our testing. Regarding the microservice-demo modification, she add some code where needed to decrypt the message to maintain the integrity of the data flow. Jonathan set up the microservices demo in the GCP cluster for benchmarking, and began working on deploying the benchmarks. Jonathan, Yingjie and Akshat all wrote and edited the project proposal, midterm report, and final report.

References

[1] gRPC. https://grpc.io/, 2020. Accessed: 2020-10-01.

[2] Protocol Buffers. https://developers.google.com/protocol-buffers, 2020. Accessed: 2020-10-01.

[3] GoogleCloudPlatform. Microservices-demo. https://github.com/GoogleCloudPlatform/microservices-demo/, 2020. Accessed: 2020-10-01.

[4] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. *ACM SIGPLAN Notices*, 51(6):631–647, 2016.

[5] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 291–304, New York, NY, USA, 2009. Association for Computing Machinery.