Secure Benchmarking in the Cloud

Axel Schroepfer, Andreas Schaad, Florian Kerschbaum
SAP Research Security & Trust
Vincenz-Priessnitz Str. 1
76131 Karlsruhe, Germany
firstname.lastname@sap.com

Heiko Boehm & Joerg Joos
Microsoft Deutschland GmbH
Konrad-Zuse-Straße 1
85716 Unterschleißheim, Germany
heikob | joerg.jooss@microsoft.com

ABSTRACT
Benchmarking is the comparison of one company’s key performance indicators (KPI) to the statistics of the same KPIs of its peer group. A KPI is a statistical quantity measuring the performance of a business process. Privacy by means of controlling access to data is of the utmost importance in benchmarking. Companies are reluctant to share their business performance data due to the risk of losing a competitive advantage or being embarrassed.

We present a cryptographic protocol for securely computing benchmarks between multiple parties and describe the technical aspects of a proof of concept implementation of SAP’s research prototype Global Benchmarking Service (GBS) on Microsoft’s cloud technology Windows Azure.

Keywords
Access Control, Privacy, Benchmarking, Cloud

1. INTRODUCTION
Benchmarking is the comparison of one company’s key performance indicators (KPI) to the statistics of the same KPIs of its peer group. A KPI is a statistical quantity measuring the performance of a business process. Examples from different company operations are make cycle time (manufacturing), cash flow (financial) and employee fluctuation rate (human resources). A peer group is a group of (usually competing) companies that are interested in comparing their KPIs based on some similarity of the companies. Examples formed along different characteristics include car manufacturers (industry sector), Fortune 500 companies in the United States (revenue and location), or airline vs. railway vs. haulage (sales market).

Privacy is of the utmost importance in benchmarking. Companies are reluctant to share their business performance data due to the risk of losing a competitive advantage or being embarrassed. As one result of our research work we presented a cryptographic protocol for securely computing benchmarks between multiple parties. We implemented the protocol in terms of a platform for secure benchmarking. The benchmarking platform is not supposed to acquire the plain text KPIs from the companies acting as a trusted third party, but rather the KPIs are to be kept entirely private to the companies. In the privacy-preserving protocol the benchmarking platform is a regular participant without any input. While the privacy protects the confidentiality of the KPIs for the companies, it alleviates the benchmarking platform from the burden of storing and handling them and protects it from the potential embarrassment due to accidental revelation.

Another important aspect of the service provider model is that the subscribed companies only communicate with the service provider, but never amongst each other. Anonymity among the subscribed companies is a required feature and can only be achieved, if they do not need to address messages to others. The precise requirement for anonymity is that subscribers do not know or refer to any static identifier of other customers (e.g. IP addresses, public keys, etc.). Any static identifier will reveal changes in the composition of the peer group to the subscribers in subsequent executions of the protocol which is undesirable and may break the privacy of the entire system. In many cases, the service provider wants to know the identity of the subscribers for billing purposes, which simplifies communication. In order to keep the proposed protocols practical, they need to be optimized in computation and communication cost. One measure is the number of rounds that are needed to complete the protocol. A round in the service provider model is a step in the protocol that all subscribers need to complete before any subscriber can initiate the next step. The proposed protocols have a constant number of rounds. Another measure is the communication complexity of the protocol. Our protocol has a constant (i.e. linear in the length of the security parameter) communication complexity for each subscriber independent of the number of subscribed companies. Our protocol (cf. [1] for details) presents from our view the first practical implementation of privacy-preserving benchmarking for statistics average, median, variance, best-in-class and maximum. It addresses a number of trade-offs in its distributed systems (single central platform) and security (key distribution and security assumptions) architecture that are tuned for practical performance and economic benefit.

1.1 SAP’s Global Benchmarking Service
We have implemented the benchmarking system of [2] in a first prototypical version in Java. The communication of the cryptographic protocol is based on objects at the service provider side that are called by stubs at the service consumer side. We use Java RMI (Remote Method Invocation) to accomplish the task. Each party of the protocol maintains a local SQL database to store inputs and outputs of the protocol. An adapter allows synchronizing the database with a SAP Business Warehouse system. Both types of parties (service consumer and provider) use a web based GUI (graphical user interface) to alter database information (e.g., settings, peer group formations, etc.). The GUIs use web services to transitorily access the database. The described setup supposes to run the mentioned components (except the database) within a web application server. The service provider stack is recommended to be installed on a computational powerful machine while a mobile device is acceptable for the client stack.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SACMAT’13, June 12–14, 2013, Amsterdam, The Netherlands.
Copyright © ACM 978-1-4503-1950-8/13/06...$15.00
2. CORE COMPONENTS

The following section describes the relevant architectural components that are involved in the construction of the cloud-based benchmark service, and their Windows Azure specific implementation.

2.1 Entities: Services, Providers & Consumers

We retain the basic platform architecture from our former prototypical construction. That is, we have a service provider (e.g., SAP) that aims to provide a service for secure benchmarking to its customers. Customers will consume the service, respectively subservices, via software running at client side. The service provider has a couple of essential tasks. He has to maintain users; maintain peer groups; schedule benchmarks; store and provide results of performed benchmarks; and act as party within the benchmark protocol in the role of the service provider.

While most of these tasks can be seen to be management tasks, we decide to group them into one service, the benchmark management service. For the lastly mentioned task, the participation in the benchmark protocol, we design an independent service. As mentioned in the beginning of the document, the protocol is constructed in a way that yields multiple sections or rounds (five, to be precise). The reason for this separation is that each round produces a partly result, i.e., one of the statistics (e.g., the median that is used in a later round in order to compute the variance). The section on scaling discusses a possible further separation of rounds into independent services. A fine-grained service separation can be extremely useful, as it allows to technically and economically optimal use the scaling infrastructure of Windows Azure.

2.2 Communication

The construction of the underlying benchmark platform bears in mind that in a practical settings clients tend to operate from within a company network that is usually protected by a firewall. Moreover, access to the outside network may be restricted by using a proxy server. That is, communication from a client’s perspective can be assumed to be limited to outbound HTTP (respectively HTTPS) traffic. For that reason the cryptic protocol is designed as a sequence of communication calls from the customer to the central service provider.

The basic messaging paradigm hinders native synchronization of clients by the service provider. Synchronization however is a vital aspect and required at several points before and within a run of the protocol. In order to accomplish the wanted behavior, we either employ a polling mechanism (consumerside-wise) that is compatible with the basic communication paradigm (i.e., a client calls a corresponding service method recurrently) or we use relative timestamps (providerside-wise).

2.3 Protocol Context - Emulating Stateful Services

Implementing the formally introduced services on Windows Azure leads straightforward to a problem: services for Azure are supposed to maintain durable state outside of each individual compute instance. This might be fully acceptable for the management service exposing methods that would be considered as static from an object oriented programming paradigm point of view. But it is an issue for the service involved in the cryptographic benchmarking protocol. Namely, as mentioned earlier, the protocol is separated into rounds. That is, a self-contained run of the protocol maintains a context, i.e., intermediate values. Again, from an object oriented programming paradigm point of view, a client’s call is comparable to a non-static (or instance) method call of an object.

The reason for this design paradigm, however, is reasonable. Services are deployed to the Windows Azure Cloud, i.e., technically this means, they are replicated over multiple virtual and physical machines. A client will always consume a service in terms of a web service call with an endpoint addressed by an internet URL. That also means, for a client, it is practically impossible to hit in two adjacent calls the identical instance of an object that is represented by the service. Hitting the identical instance would also collide with paradigms of robustness and scalability, like availability through replication, load balancing, etc.

We achieve the necessary protocol context by using Blob storage technology of Windows Azure. Blob storages can be thought of as byte arrays (Blob stands for binary large object) that are addressed via an internet URL. A service (respectively, a method) can read or write the uniquely named Blob. At the point of writing this document, the API support by the Windows Azure SDK (software development kit) for locking a Blob (in order to achieve exclusive access) was not available yet (but planned for a future release). We tackle the point in the following way. We introduce a Blob that is created on protocol start and gets attached to a default name the ID of the protocol run (which is chosen randomly at the start). This is the so called “roundall” Blob. Since the cryptographic protocol is well structured, there is a determined next round and protocol party. We store this information in the Blob (i.e., the first party and the first round on initialization).
2.4 Data Management

improvement. Replicating segments of the (benchmark) context (to round protocol run into Blobs for each round (round1, …, round5). Blobs as well. For instance, we separate the full context of a lock mechanism, we can handle read and write access to other is the last party, increment the round id. Using this emulation of a update the Blob. Precisely, it will increment the party id, and if it is the last party, increment the round id. Using this emulation of a lock mechanism, we can handle read and write access to other Blobs as well. For instance, we separate the full context of a protocol run into Blobs for each round (round1, …, round5). Replicating segments of the (benchmark) context (to round contexts) is supposed to result in additional performance improvement.

2.4 Data Management

Both technical entities, the service provider and the service consumer, require data persistence. While the consumer will store its private data either encrypted in the cloud or (potentially not) encrypted locally, the service provider will store by design its data in the cloud. The following sections describes, how this is technically achieved using most noticeably Windows Azure Technology.

2.4.1 Consumer Side

The consumer stores private information. In particular, he stores his private KPI input that is to be benchmarked, as well as results of executed benchmarks. We therefore see a local storage in a SQL database to be the most realistic setup for a practical application. However, it is also possible to let the data be stored in the cloud by using Windows Azure SQL Database. While encryption for local storage does not necessarily need to be handled by the application itself (either because there is no local encryption or the database system encapsulates it), care must be taken when data is stored in the cloud. The application then requires according setup and handling in the code respectively, in order to assure proper encryption (respectively local decryption).

For this prototype, we stick with a local database system running Microsoft SQL Server because we last but not least do reuse parts of the Adobe Flash based GUI of the former (not cloud enabled) version of the prototype. It reads data transitively via an SQL connection from a local database. Moreover, this setup fosters the implemented integration of an SAP Business Warehouse system, providing aggregated KPIs.

2.4.2 Provider Side

The service provider similarly requires data persistency in order to maintain user registrations and peergroup settings. Moreover, the service provider is supposed to store the results of performed benchmarks. This is in order to let the service provider make this information available to other customers (e.g., to newly joined customers by a purchase or benefit). Intuitively, the service provider data is not as confidential as the private KPIs of customers. However, analogously to customers, the service provider may perfectly (want to) store data in encrypted form using encrypted storage services of Windows Azure. For the sake of this prototype, we see the authorized access to unencrypted data to be sufficient. We therefore save all service provider data in the cloud using a Windows Azure SQL database.

2.5 Third Party Identities

The global benchmarking service contains a list of registered users in his database. The registration is required to map users to peergroups and benchmarks in order to maintain a proper scheduling, as well as to be able to charge customers appropriately. Identity is also important when it comes directly to an execution of a benchmark. It is vital to make sure that the supposed parties are indeed the once participating in the benchmark run, i.e., those inputting data and learning the statistics as result. There are many different ways to establish these required authenticated message channels. The foundation of all of them is identity. It is cumbersome for service consumers as well as for the service provider to maintain an identity. This is true for the registration as well as for the actual application. For this reason, techniques like single sign on have been constructed which try to hide the complexity of maintaining the identity of a communication or session from users by running standardized protocols in the background. Such protocols are SAML (Security Assertion Markup Language), XACML (eXtensible Access Control Markup Language) and WS-* (web service security protocols). Windows Azure understands these protocols, hence is able to participate as an entity in such protocols. That is what enables the usage of third party identity providers. Technically, a third party will issue a (signed) token which rules out a user as a registered user of the third party. An email provider can for instance provide a (signed temporary valid) token that a particular user is actually known to him. In addition the token may provide further information, e.g., the name or email address or facts like that the respective user is logged onto the system. The information (e.g., the name) is, again, standardized. The pieces of information that are provided depend on the respective third party.

For our benchmarking application, we required at least a unique identifier (e.g., a constant random string). As an example, we set up our cloud application to accept tokens issued by the Windows Live ID service that provides a constant random string as unique identifier. If the service consumer is already logged into Windows Live ID (e.g., by his messenger), then the application is able to directly start to use his identity (a browser component integrated in the application therefore accesses a corresponding cookie). Otherwise, a login screen will appear in order to log into Windows Live ID (and to set the cookie). The final technical effect is, that a method call by the client stub will let the service method of the service provider be able to access the token (hence the unique identifier) of the caller. Within the method, this can be mapped to the service consumer identifier in the service provider database.
involved in an upcoming benchmark. The method of the service is called by the clients, e.g., to figure out whether he is the benchmark management service. The benchmark management service has successfully deployed cloud application. We use this API within the service management and thus the configuration of the (representational state transfer) based API that provides access to deployed and in line. This task is accomplished by a REST API. The construction incorporates the fact, that a change to the configuration will take a certain number of seconds to take effect. Based on the benchmark schedule, the number of participants, the time for up-scaling (respectively down-scaling) and a simplified economic framework (service charges, resource consumption/payment), a dynamic scaling for the benchmarking service has successfully been implemented.

4. CONCLUSION
We successfully implemented a prototype of our benchmarking system for the cloud using Microsoft Windows Azure as platform. We adapted the architecture of our first prototypical (Java based) implementation. This has been done just punctually for the consumer side and in broader sense for the provider side. Although the Windows Azure platform would have been able to handle our former Java implementation of the cryptographic protocol, we rather decided to re-implement the cryptographic protocol and respective cryptographic tools in C# and Microsoft .NET. We did so in order to be able to natively access existing API based features of the platform (e.g., scaling). Figure 2 shows the final architecture of our cloud enabled prototype of the secure benchmarking system.

We have the consumer that runs a local database to store his private KPI values that he possibly received from a connected SAP Business Warehouse. He takes part in the cryptographic protocol for secure benchmarking by calling cloud service methods of the service provider. That is done within a MFC (Microsoft Foundation Class) client util. The consumer keeps running a local (tiny) web application server that provides a GUI web service (i.e., a web service that renders the benchmark results from the local database into a format understandable for the GUI) and that hosts the Flash GUI (as depicted in Figure 3).

3. PERFORMANCE – SCALING ON DEMAND
A Windows Azure subscription includes a set of resources (e.g., number of CPU cores) the subscriber can consume. It is up to the subscriber to set the scaling of his cloud application accordingly. The subscriber for instance could decide to double the computational power of his cloud service. He could also decide to only let the service run half of each day to stay with no additional charging. The set of resources is grouped by instances\(^1\), ranging from XS (extra small) to XL (extra-large) and defines contained resources like number of CPU cores, number of SQL databases, number of Storage transaction and the like. Windows Azure offers multiple ways to set the instances of a cloud application (to be precise, the instance type of a role), e.g., static or dynamic setting.

3.1 Static Scaling
Naturally, the Windows Azure SDK allows choosing the instance type as an option within the development settings. The deployment utility reads the option and sets the cloud environment accordingly on deployment. This way of setting the instance type is appropriate for cloud applications that know upfront what their demand on resources will be, respectively their economic bounds and that these will not change much over time. Here, static scaling is sufficient.

3.2 Dynamic Scaling
In the case of benchmarking, however, the demand for resources varies. Precisely, it depends on the number of participants. In particular, cryptographic operations, that have a high demand for CPU, are often proportional to the number of participants according to the protocol description in [1]. A valuable feature in that respect is that Windows Azure allows (next to scaling at deployment) also a scaling when the application has already been deployed and is online. This task is accomplished by a REST (representational state transfer) based API that provides access to the service management and thus the configuration of the successfully deployed cloud application. We use this API within the benchmark management service. The benchmark management service is called by the clients, e.g., to figure out whether he is involved in an upcoming benchmark. The method of the service performs a check on the number of all upcoming benchmarks, including a list of participants. Based on these numbers, a dynamic scale-up is performed by calling the formerly mentioned REST API. The construction incorporates the fact, that a change to the configuration will take a certain number of seconds to take effect. Based on the benchmark schedule, the number of participants, the time for up-scaling (respectively down-scaling) and a simplified economic framework (service charges, resource consumption/payment), a dynamic scaling for the benchmarking service has successfully been implemented.

REFERENCES

---