SSICLOPS

Extension of Security Architecture to Hybrid (Inter-) Cloud Systems

SSICLOPS Deliverable D2.3

Marios Choudary,[5] Sergiu Costea,[5] Felix Eberhardt,[2] Seppo Heikkilä,[3] Jens Hiller,[4] Oliver Hohlfeld,[4] Stefan Klauck,[2] Ville Lindfors,[1] Tapio Niemi,[3] Max Plauth,[2] Andreas Polze,[2] Costin Raiciu,[5] Matthias Uflacker,[2] and Klaus Wehrle[4]

[1]*F-Secure OYJ*
[2]*Hasso Plattner Institute for Software Systems Engineering*
[3]*Helsinki Institute of Physics*
[4]*RWTH Aachen University*
[5]*University Politehnica of Bucharest*

January 31, 2017

This report has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). It reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

Contents

Executive Summary

Cloud computing gives customers manifold benefits such as resource elasticity, virtually unlimited processing power, outsourcing of management tasks, and high availability. However, moving data storage and processing out of local server rooms to cloud systems run by third parties also introduces new challenges. As a consequence, many data cannot be outsourced to the cloud today, e.g., due to legal reasons. The main concerns regarding cloud usage are related to privacy and security requirements that cannot sufficiently be addressed by the cloud today.

Using the cloud, potentially privacy sensitive data leaves the domain of the data owner and is handled by third party cloud providers. While private users and companies need to accept the policy established by cloud providers to use the cloud, the opposite direction, i.e., a specification of expectations on the handling of data by cloud users, is not possible today. We present the evaluation of our previously defined *Compact Privacy Policy Langauge* (CPPL) which enables cloud customers to attach their expectations as per-data item policies to each data item that they transmit to the cloud. As we show, CPPL – being designed with the special requirements of cloud environments on policy use in mind – enables efficient and automated use of such per-data item policies in cloud environments. Furthermore, especially in federated clouds, i.e., clouds that consist of multiple interconnected clouds of different cloud providers, CPPL can be used during bootstrapping of new application instances to ensure the availability of resources as requested by users. In this document, we describe the benefits of CPPL in cloud and federated cloud settings, show CPPL's suitability for these environments with an extensive evaluation. Furthermore we provide in-depth discussions about how CPPL can be integrated and leveraged in the context of the SSICLOPS use cases.

Given the possibility for cloud users to express their expectations via policies as per-data item annotations, data security is still at risk during communication between cloud user and the cloud. This communication faces multiple security risks and should even provide protection against nation-scale attackers. It further exacerbates for federated cloud computing as data may be repeatedly transferred between clouds of different cloud providers. To also secure data on transit to or between multiple clouds in a federated cloud, we present our work on a new hardened transport security protocol that leverages multipath communication. Combining this secure transit of data between customers and cloud providers as well as between clouds with the mechanism of an efficient expression of user expectations on handling of data by cloud providers will enable customers to use the cloud even for highly privacy sensitive and security relevant data.

1. Introduction

Interconnections of several private clouds in federated or inter-cloud scenarios create new challenges for data privacy, data protection and data security. Therefore, as part of SSICLOPS, we have to deal with all these aspects. First, as will be detailed in the following sections, we have considered the privacy requirements of users. Then, we have designed protocols for secure data transfer. Finally, we have integrated the key results into different scenarios and implemented mechanisms for facilitating data protection.

Today, cloud providers treat data of customers according to privacy policy statements which they tailor to their own needs. Lacking a channel in the opposite direction, users are unable to specify their own requirements regarding the handling of their data that they provide to cloud services [13, 14]. Consequently, users only have the choice to either accept the data handling by the cloud provider as specified in the privacy statement or to refrain from the usage of the cloud service. Additionally, cloud providers typically provide only a single privacy statement which specifies the handling procedures for all data. This neglects the different levels of privacy required for different data, e.g., banking information or social security numbers require significant efforts with respect to privacy handling while other data such as the employer of a person can be less privacy sensitive. However, with a single privacy statement, all data is handled the same way, either ignoring the increased demands of very privacy sensitive data or waisting resources for protection mechanisms for data that does not need it. These problems are not limited to private users, but also affect business users, e.g., 55% of IT professionals rank the lack of control over data in public clouds as one of their top three concerns [17]. Furthermore, 78% need to comply with regulatory mandates and 57% of those refrain from storing data that is affected by these regulations in private or public clouds [17]. The ability to control where data is stored and processed as well as the ability to compare security levels of different cloud providers would improve their confidence to make use of the cloud [17]. Thus, the ability of users to specify their own expectations on the data handling by cloud services, as well as the handling of data based on the individual requirements of this data is inevitable for future cloud environments that handle privacy sensitive data [13, 14].

Our approach to enable an automatic handling of user expectations employs a privacy policy language that enables users to specify their expectations on the data handling individually for each data item. In Deliverable 2.1 [11], we provided a detailed requirements analysis for such a privacy policy language. We derived that a policy for a data item should travel with this data as a small data annotation [13, 14] that allows evaluation of the policy whenever required, e.g., when selecting a storage place for data, decrypting data for processing, or even determining allowed routing paths between data centers. Furthermore, we analyzed existing (privacy) policy languages with respect to their applicability to the cloud scenario. This analysis showed shortcomings for all of these analyzed policy languages in this setting. In Deliverable 2.2 [10], we presented our design of a privacy policy language that fulfills the derived requirements for the application in the cloud scenario. Furthermore, we discussed strategies for implementing and integrating policy concepts into established software components, e.g., Hyrise in-memory database instances in an *OpenStack*-based cloud environment.

In this deliverable, we extend the description of policy use from single cloud provider scenarios to the employment of privacy policies in federated clouds based on the concepts presented in Deliverable 2.2[10]. Furthermore, we discuss the performance of our privacy policy language CPPL based on an extensive evaluation. Then, we present the design of new multipath security protocols that are suitable for federated clouds. Finally, we describe the envisioned use of our privacy policy language in the use cases of the SSICLOPS project, i.e., we outline the use cases and give an in depth analysis of the benefits that come with the support of considering user expectations when handling of requests and data.

The structure of this deliverable is as follows: In Section 2, we outline the use of privacy policies in federated clouds, i.e., we extend our former description of the single cloud provider scenario to that of a cloud built up from resources of multiple cloud providers. Following, we give a short recap of the design principles of our *Compact Privacy Policy Language* (CPPL) and discuss its performance based on synthetic benchmarks, comparison to related work, and initial use case evaluation in a local testbed in Section 3. Then, in Section 4, we present our new security protocols based on multipath communications and suitable for use within federated clouds. In Section 5, we emphasize the benefits of privacy policy support for cloud computing by providing an in-depth analysis of the use of privacy policies in the SSICLOPS use cases comprising an OpenStack testbed, in-memory databases (Hyrise), High-Energy Physics workloads, Network Function Virtualization by Internet Service Providers, and analysis of user supplied data for security threats. Finally, we conclude the deliverable in Section 6.

2. Policy Languages in Federated Clouds

In the previous deliverables [10, 11] we mainly focused on the scenario of single cloud providers. More specifically, while we always kept in mind the requirement to extend to a federated cloud during our analysis of existing policy languages and the design of our compact privacy policy language (CPPL) [15], we never explicitly outlined the use of privacy policy languages in federated clouds. In this section, we extend our description of the single provider cloud scenario to the federated cloud scenarios where the cloud consists of multiple smaller private clouds managed by different cloud providers. In this scenario, the use of privacy policy languages enables automated negotiation regarding handling of data between the cloud providers, e.g., operational data of the services itself but also user data.

The initial motivation for the use of privacy policy languages to give users control over the handling of their data by the cloud does not change when extending the scenario to federated clouds. Thus, before we extend our scenario to federated clouds, we recap the usage scenario of single cloud providers already discussed in the previous deliverables [10, 11]. Afterwards, we analyze the extended possibilities enabled by privacy policy languages in the area of federated clouds.

2.1. Single Cloud Provider Scenario

The use of cloud services by private users as well as enterprises comes at the cost of handing over privacy sensitive user data to cloud providers. Thereby, cloud providers handle the supplied data based on privacy policies which typically are static statements dictated by the cloud provider. Consequently, users do not have the possibility to negotiate the applied privacy rules with the cloud provider, i.e., users have to accept the privacy policy of the cloud provider or refrain from usage of the service. If users still decide to use the service and hand over their data to the cloud provider, they lose control of their data in the range specified by the privacy policy [13, 14]. Furthermore, users are not only unable to specify their own requirements for the handling of their data. Instead, privacy statements employed by cloud providers to communicate their privacy policies to users are legal documents that are often hard to read and understand. As a consequence, users often do not read the legal statements, and – even worse – mistakenly assume data storage and handling in the cloud provides the same privacy guarantees as data handling at their local devices [18]. Hence, users hand over their data to cloud systems staying oblivious regarding the actual handling of data by cloud providers.

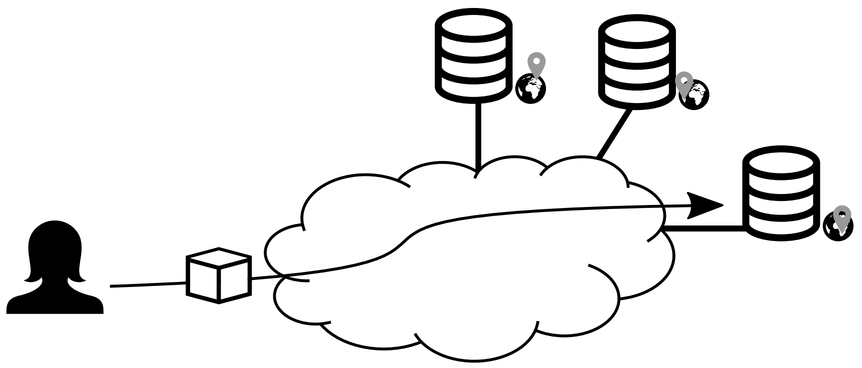

Figure 1: Data handling in a traditional cloud: Data is handled by a single cloud provider.

However, there are plenty of different expectations that users would like to express to stay in control of the handling of their data that is transferred to the cloud [13, 14]. As an example, Figure 1 shows a user that selects the storage location of the database that handles the cloud data. Other expectations are the selection of encryption methods, kind of storage, guaranteed data deletion, access logging or offered redundancy [11, 18]. The formulation of such expectations in privacy policy languages enables cloud providers to consider the various different privacy expectations of different users and even allows them to apply rules that are specific for a single data item. As expectations based on policy languages also provide the ability for automatic processing and complying with user expectations, they are appropriate for the usage in the cloud scenario [13, 14].

However, corresponding mechanisms must be highly scalable to cope with the huge amount of data. More specifically, supporting possibly different expectations for each data item requires this expectation to be available whenever an action is performed with the data, e.g., determining the storage location of data or retrieving it for processing. In order to ensure the availability of the expectation for these checks, the user expectations can be attached to the corresponding data item as data annotation [13, 14]. The primary requirements for a privacy policy language that supports the data annotation mechanism and is suitable for the cloud scenario are (i) a small storage footprint, (ii) human readability, (iii) detection of conflicts at time of specification, (iv) sufficient evaluation performance, (v) adequate expressiveness for the cloud scenario, (vi) extensibility for adaptation to new policy statements that come up with emerging cloud services, and (vii) a matching mechanism that checks if the offer of a cloud provider fulfills the expectations formulated in the policy [11].

2.2. Extension to Federated Cloud Environment

So far, we mainly focused on the usage of policy languages in cloud environments with a single provider. Extending this scenario to a federated cloud environment does not fundamentally change this scenario but adds up new possibilities opportunities for cloud providers. While,

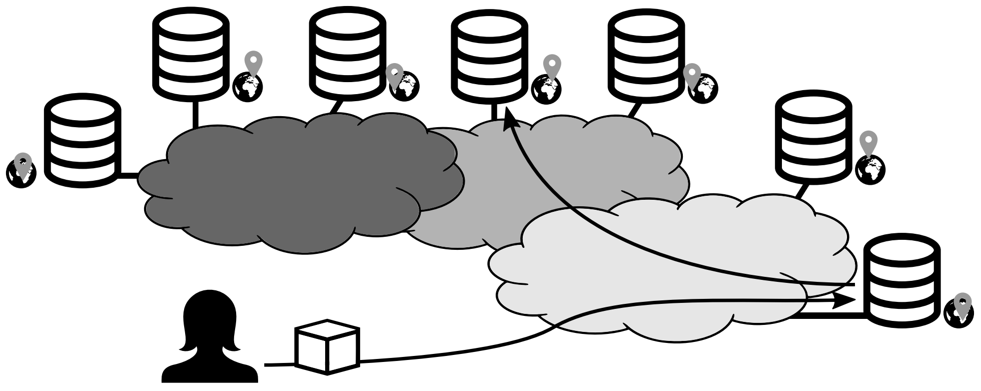
Figure 2: Data handling in a federated cloud: User data may be handled by multiple cloud providers of the federation.

the communication between user (comprising private users as well as companies) and cloud provider remains unchanged, data can be forwarded and handled by other cloud providers within the federation as depicted in Figure 2. In this federated cloud scenario, per-data item privacy policy annotations easily allow for adherence to user expectations in the whole federation as they travel with data across the cloud boundaries. Furthermore, when cloud providers add up their own expectations to that of the users, the data annotations can also be employed to easily add up further restrictions by the original cloud provider. Thus, any data handling policy can travel within the data annotation without any need for further negotiation of such policies between cloud providers of the federation.

In the following, we especially focus on two new aspects that emerge with a federated cloud. First, a federated cloud increases the availability of resources for each cloud provider in the federation. Thereby, it also increases the likelihood that, if a cloud provider cannot handle data with his own resources due to the specified user expectations, another cloud provider in the federation has suitable resources to handle the data. To leverage this opportunity, the federation requires a mechanism to find suitable resources for specified user expectations. Second, if there are still not enough resources available in a federated cloud to handle data with often occurring user expectations, cloud providers may even negotiate to create new resources to enable the handling of corresponding data. For example, a federation may launch new instances of in-memory databases if many users request fast access to their data.

2.2.1. Negotiation of Resource Usage

In a federated cloud, the decision why and when data is not handled by the original cloud provider but by another cloud provider in the federation may be complex and is determined by many factors, e.g., cost, quality of service guarantees, load factor, or the availability of resources for fulfilling a requested policy. For simplicity, we assume that a cloud provider handles data on his own systems if a sufficient amount of resources is available that can adhere to the policy specified for the data item that needs to be handled. Consequently, cloud providers only utilize resources of the federation if own resources are not sufficiently available. We note that we take this assumption for the sake of simplicity in our descriptions and any other strategy could be similarly applied.

In the single provider scenario, the cloud provider has to reject the handling of data if no resources are available that adhere to the corresponding policy. In a federated cloud, however, the cloud provider can check with other providers of the federation if they have resources available that can process this data item. Instead of negotiating availability of suitable resources with each private cloud in the federation, such requests could be handled by a *Federation Policy Decision Point* (FPDP) that obtains information about available resources and corresponding properties of the private clouds and can therefore decide if another cloud provider can handle the data. This way, cloud providers in the federation can not only profit from the elasticity of resources but also may increase the set of supported user expectations and therefore handle data whose policy does not allow them to process the data in the single cloud provider scenario.

2.2.2. Policy-Aware Creation of Resources

Beyond the mere usage of existing resources, the federation could also track requests that can not be fulfilled due to unfulfilled policies and react accordingly, e.g., by bootstrapping new instances of a service at a specific location of the federated cloud. For example, if users often request the use of in-memory databases, the federation could automatically use available resources to bootstrap another in-memory database instance. Such a flexible utilization of resources would enable the federation to adapt to the needs of customers which are expressed with the policies. For users, this enables the use of the cloud while data is handled according to their needs while cloud providers are able to offer their service to privacy-aware users or customers that depend on the ability to restrict usage of data, e.g., due to legal conditions. However, this requires to identify often used policies that cannot be fulfilled. With the FPDP (cf. Section 2.2.1) there exists an instance that obtains knowledge of such policies. It could therefore create statistics and suggest the creation of resources accordingly.

3. SSICLOPS Policy Language CPPL

Existing policy languages do not sufficiently support the requirements for their usage in cloud and federated cloud environments [11, 15]. For an extensive analysis of related work as well as past and ongoing related EU projects we refer to [11, 15]. To enable the efficient use of policies in cloud and federated cloud environments, we developed the Compact Privacy Policy Language (CPPL) [15]. Using this policy language to create per-data item annotations, cloud providers and cloud federations can realize automated adherence to customer expectations regarding resource usage as well as use it for inter-cloud negotiation of resource creation and configuration (cf. Section 2). Addressing the requirements of policy use in clouds and federated clouds, CPPL features expressiveness and extendibility, but at the same time addresses the mostly neglected requirements of runtime performance as well as communication and storage efficiency. In this section, we discuss the evaluation results of CPPL which show its suitability for the use in cloud and federated cloud environments.

3.1. Concepts

Before we discuss the evaluation results of CPPL, we shortly introduce the concepts of CPPL that allow for expressiveness and extendibility but also good runtime performance as well as efficiency regarding communication and storage overhead. We refer to [10, 15] for more detailed descriptions of theses concepts.

In CPPL, we use domain parameters to enable an expressible and extensible policy language that also achieves considerable compression gains and provides efficient matching. To this end, the domain parameters specify the variables, functions, variable types, and relation types that can be used in a policy based on these specific domain parameters. Thus, extending the domain parameters extends the expressiveness of corresponding policies. However, at the same time, the domain parameters restrict the policies and therefore allow for efficient compression by, e.g., efficiently encoding variables and functions with reference numbers.

Furthermore, we use *redundancies* to decrease the footprint, i.e., if a variable value occurs multiple times in a policy, we encode the actual value only one time and use footprint efficient references to this existing encoding. The same holds for redundant relations which can be referenced similarly yielding even bigger compression gains compared to redundant variables.

Moreover, the domain parameters specify the type of each variable, e.g., Boolean, 32 bit integer, or string. Considering integers of different sizes, e.g., 8 bit integers, 32 bit integers and 64 bit integers, these require different sizes for encoding. However, when the actual value of a 32 bit integer variable is representable with an 8 bit encoding, the value with 32 bit unnecessarily increases the footprint of the policy. Instead, we *downsize integers* to efficiently encode small values. More specifically, a variable may support integers of, e.g., up to 64 bits. However, when the actual value is representable with an integer type that requires less bits for encoding, the CPPL compression encodes the value with this smaller integer type.

The described compression methods enable efficient transmission and storage of policies. Nevertheless, to match the policy with properties of a cloud system, standard compression methods would require decompression before this matching step. With CPPL, however, we support a direct matching of policy and cloud system properties without explicit decompression. Instead, the matching algorithm operates on the compressed data directly which significantly saves runtime during the matching process. Furthermore, we cache the result for each relation and reuse it for redundant relations to save corresponding runtime during policy matching.

3.2. Evaluation Results

To show the applicability of CPPL concepts and our implementation, we performed extensive evaluations. These comprise synthetic evaluation to get an impression on the effect of changes to the different inputs, i.e., policy and domain parameters. A further analysis of CPPL based on policies already used in the real world shows its improvements over existing privacy policy languages. Finally, we show the applicability of CPPL. To this end, we analyze footprint overheads when annotating real world IoT data and extend this by a runtime performance analysis for policy-aware machine learning on CPPL-annotated real world data. We initially published these results in [15].

3.2.1. Synthetic Evaluation

Performance and scalability of CPPL depend on the size of the policy and the comprehensiveness of the domain parameters. To quantify the impact of these inputs, we performed synthetic benchmarks in a local testbed on a machine equipped with desktop-grade hardware (Intel i7 870, 4 GB RAM, Ubuntu 14.04). We performed 100 runs for each measurement point and depict mean values with 99% confidence intervals. In the results, we neglect the overhead for initialization which is in the order of 1.2 ms for both compression and matching. This is reasonable as, e.g., the domain parameters specification will typically be known beforehand and can thus be loaded and parsed during system boot.

3.2.1.1. Increasing Policy Size

To quantify the influence of the size of policies on the storage footprint, compression runtime and matching runtime, we performed measurements with a fixed set of domain parameters that

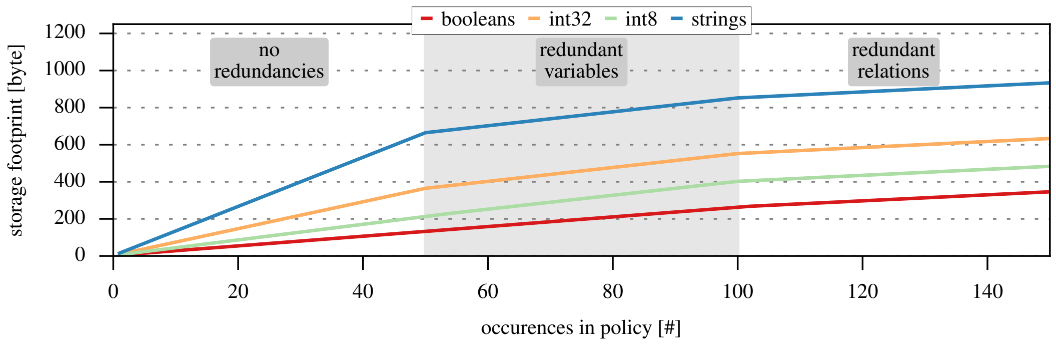

Figure 3: **Policy size vs. storage footprint.** The compression gain is significantly improved by redundant variables and relations. Furthermore, also integer optimization improves the compression.

support 100 Boolean, integer, and string variables, each. The variable policies use up to 150 relations of the same variable type. Thereby, each relation can take up to 50 (integer, strings) respectively 2 (Boolean) actual values.

We use these supported variables to analyze three different scenarios of policy compression: (i) if no redundancies occur in the policy, we cannot leverage our redundancy mechanism for compression, (ii) actual variables can occur repeatedly in a policy which we can leverage for our compression, and (iii) policies can even contain redundant relations of which all but one can be replaced by references. To study the effect of these scenarios, we first evaluate policies without any redundancies (Relations 1 to 50). We then show the effect of redundant variables by repeating already used variable values without repeating the whole relation (Relations 51 to 100). Finally, we extend this by redundant relations by reusing the first 50 relations (Relations 101 to 150). Furthermore, to evaluate the effect of CPPL's integer downsizing during compression, we use two integer sizes. More specifically, the variables specified in the domain parameters always use 32 bit, but in the policy we also employ values that can be represented with 8 bit.

Figure 3 depicts the storage footprint of a compressed CPPL policy, i.e., the amount of data that must be transmitted and stored whenever this policy is attached to a data item. Without redundancies, the size of the compressed CPPL policy scales linearly, e.g., increasing from 9 byte for 1 relation to 364 byte for 50 relations when using 32 bit integers in the policy. With redundant variables, the CPPL compression gain increases for strings (ration of 3.53) and 32 bit integers (ration 1.93). However, the identifier used to encode references consumes 8 bit. Thus, variable values that already require less than 8 bit for encoding like 8 bit integers or Booleans do not benefit from the redundant variable mechanism. When policies also contain redundant relations, CPPL further increases the compression gain, e.g., by a ratio of 2.31 for 32 bit. In contrast to redundant variables, the compression gain acquired by using redundant relations is present for all variable types. Considering the effect of integer downsizing, integers that can be represented by 8 bits (although corresponding variables in the domain parameters are specified as 32 bit integer variables) considerably decrease the storage footprint compared to integers that require 32 bit

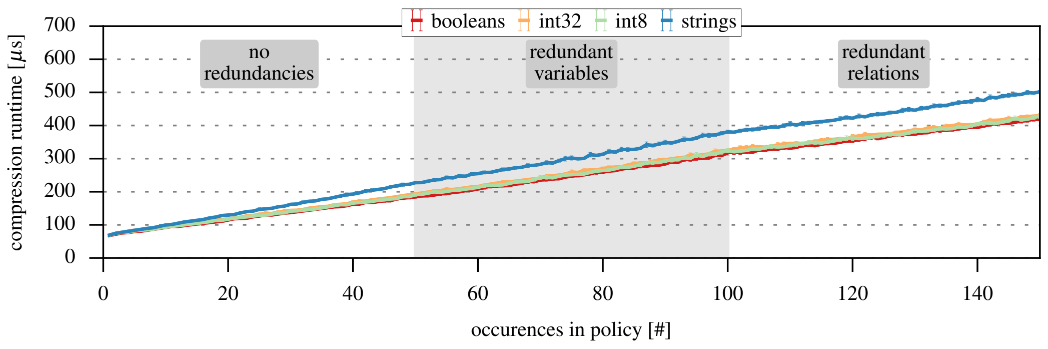

Figure 4: **Policy size vs. compression time.** Runtime scales linearly and variables with larger size increase the runtime. Redundancies as well as integer optimization have no significant impact.

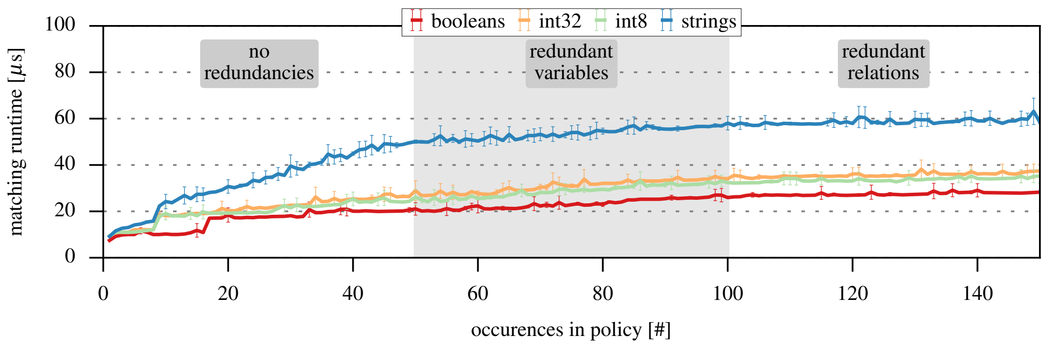

Figure 5: **Policy size vs. matching time.** The runtime increases with larger policies. Redundancies decrease the impact of the policy size. This effect increases with increasing size of the variable values such that especially policies with strings benefit.

for their representation. Thus, the integer downsizing mechanism also proofs valuable for the storage footprint.

Next, we analyze the compression runtime, i.e., the time that is required for using domain parameters to compress a textual CPPL policy. Notably, this compression step is required only once and the compressed CPPL policy can afterwards be used for multiple data items. That is, the compressed policy is independent from the actual data item that it annotates as the compression only depends on the textual policy and the domain parameters. Figure 4 shows the compression runtime depending on the policy size. The compression runtime scales linearly for an increasing policy size and increases from 68 μs for 1 relation to 418–431 μs (502 μs for strings) for 150 relations. Thus, our CPPL implementation is able to compress 1 993 to 14 754 policies per second. Comparing the different variable types, strings introduce slightly more runtime overhead as they require more time for encoding and for the comparison-based search for redundancies. While redundancies significantly improve the storage footprint, they do not noticeable influence the runtime for compression.

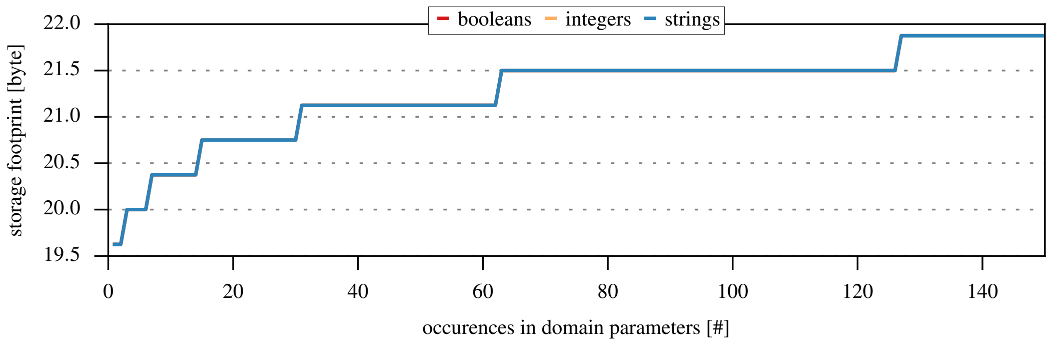

Figure 6: **Domain parameter size vs. storage footprint.** An increasing policy expressiveness increases the storage footprint logarithmically.

Finally, we present the matching runtime, i.e., the time required for matching a compressed CPPL policy against properties of a cloud system. This matching takes place whenever data is handled by cloud instances and therefore it is critical to minimize it. Figure 5 shows the results of our measurements. Compared to compression, matching is significantly faster and for larger policies an increasing policy size results in a linear increase of the runtime. When the policy does not contain any redundancies, the matching runtime for strings increases from 9 μs (7 μs) for 1 relation to 50 μs (21 μs) for 50 relations. The runtime for Booleans increases from 7 μs for 1 relation to 21 μs for 50 relations while integers require slightly more runtime than Booleans. Considering the effect of redundancies, especially the processing for strings considerably decreases such that in our example matching requires only 58 μs to match the policy containing 150 relations. Also redundancies of other variable types positively affect the matching runtime, e.g., matching the policy with 150 Boolean variables requires only 28 μs due to the introduced redundancies. Overall, the matching runtime of CPPL allows for processing of 17 126 to 134 048 policies per second.

3.2.1.2. More Comprehensive Domain Parameters

So far we only considered the impact of an increasing policy size. However, also the comprehensiveness of the domain parameters has an influence on storage footprint, compression runtime and time required for matching. Thereby, more comprehensive domain parameters enable the use of a larger variety of variables and functions in the policy. To evaluate the domain parameter impact, we use a constant CPPL policy that consists of 1 Boolean, 1 integer and 1 string relation. We then increase the number of domain parameters from 1 up to 150 for each of these variable types. Notably, redundancies and integer downsizing only depends on the textual policy which we keep fixed in this analysis. Therefore, redundancies and integer downsizing do not pose an impact and we only depict results for 32 bit integers omitting the use of 8 bit integers.

Figure 6 shows the storage footprint for increasing domain parameters. Notably, all variable types exhibit the same behavior such that all three lines lie on top of each other. With an

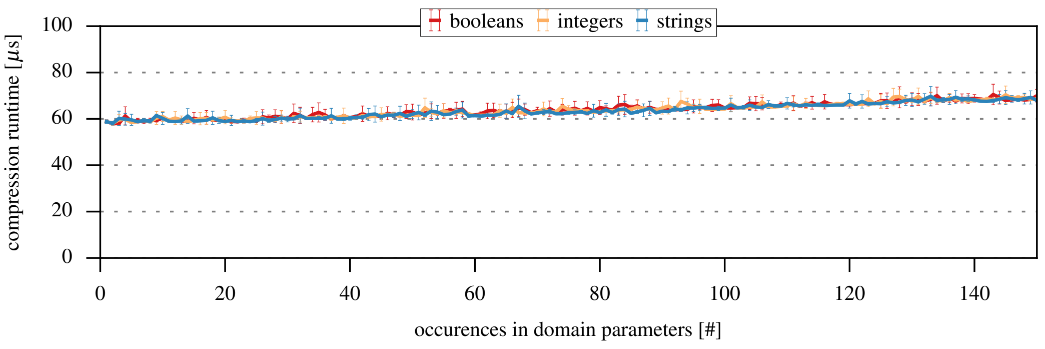

Figure 7: **Domain parameter size vs. compression time.** Increasing domain parameters comprehensiveness results in a slight linear increase of compression runtimes.

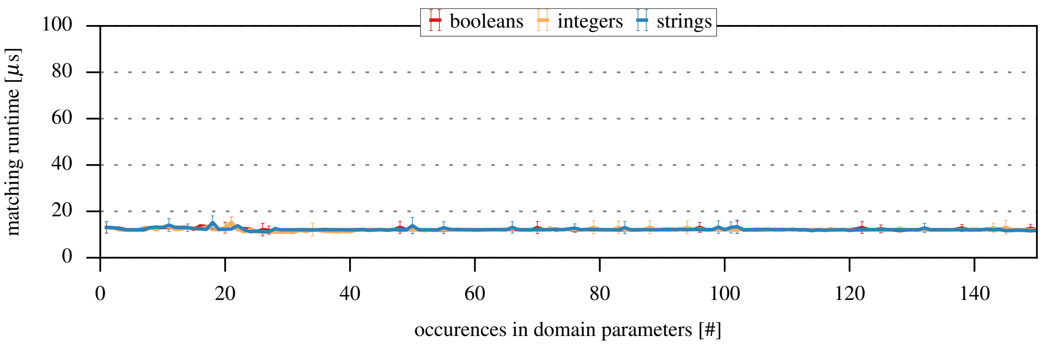

Figure 8: **Domain parameter size vs. matching time.** The very low runtime required for matching stays constant for changing comprehensiveness of the domain parameters.

increasing number of available variables and functions, CPPL requires more bits to encode variable identifiers in the compressed CPPL policy, which is independent of the variable type. More specifically, domain parameters that specify n variables require $\lceil \log_2(n) \rceil$ bits to encode the variable identifier in the compressed policy. In terms of numbers, the storage footprint increases from 19.63 bytes for 1 defined variable to 21.88 byte for 150 variable definitions.

We depict the influence of more comprehensive domain parameters on the compression runtime in Figure 7. With an increasing number of available variables, the compression runtime slightly increases linearly from 59 μs for 1 variable to 70 μs for 150 variables. Compared to the influence of the policy size this increase is negligible and approximately 14 288 to 17 004 policies can be compressed per second

Finally, Figure 8 shows that there is almost no impact of the domain parameters on the matching runtime regardless of the variable type. Here, matching always requires 12 to 13 μs which corresponds to 76 453 to 85 251 policy matchings per second. This is an important result for the cloud use case as even very complex domain parameters do not negatively affect the runtime of

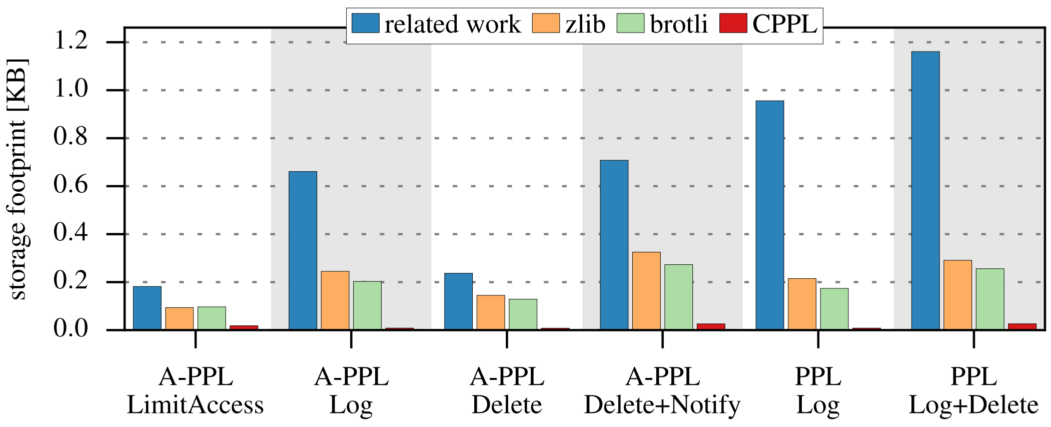

Figure 9: **Real-world storage footprint comparison with related work.** CPPL significantly reduces the storage footprint required for policy representation compared to related work.

matching which is the most often required type of processing when using data annotations to support user expectations.

3.2.2. Evaluation with Real World Policies

In the previous section, we showed the impact of policy size and domain parameter comprehensiveness on storage footprint as well as compression and matching runtime. Now we shed light in the real world applicability of CPPL regarding storage footprint and matching performance.

First, we compare the storage footprint of policies expressed in policy languages from related work with their corresponding representations as compressed CPPL policies. More specifically, we use six XML-based policies, namely four A-PPL policies (one limiting access based on location, purpose, and time conditions [2]; one logging access, deletion, and sent operations; one specifying a deletion date; and one defining a deletion date and notification on deletion [7]) and two PPL policies (one specifying logging of three different actions and one extending the former with a deletion date [33]). We also compare CPPL's compression gain with that achieved by generic compression algorithms, namely *zlib* and *brotli*.

The results of this analysis are depicted in Figure 9. It shows that CPPL significantly outperforms related work with respect to storage footprint, e.g., it is able to reduce *A-PPL LimitAccess* from 182 byte to 18.25 byte and *PPL Log* from 956 byte to only 8.5 byte. Even when considering generic compression algorithms, CPPL provides much smaller representations of policies. For large policies, zlib and brotli achieve a compression ratio of 2.18 up to 5.49, but CPPL reduces the size by a ratio of 27.10 up to 112.47. For smaller policies, zlib and brotli even perform worse, achieving a compression ratio of only 1.63 up to 1.94. Here, CPPL still achieves a reduction by a ratio of 9.97 up to 29.63. Thus, CPPL enables an enormous reduction of the overall required storage space and communication overhead as we further detail in Section 3.2.3.

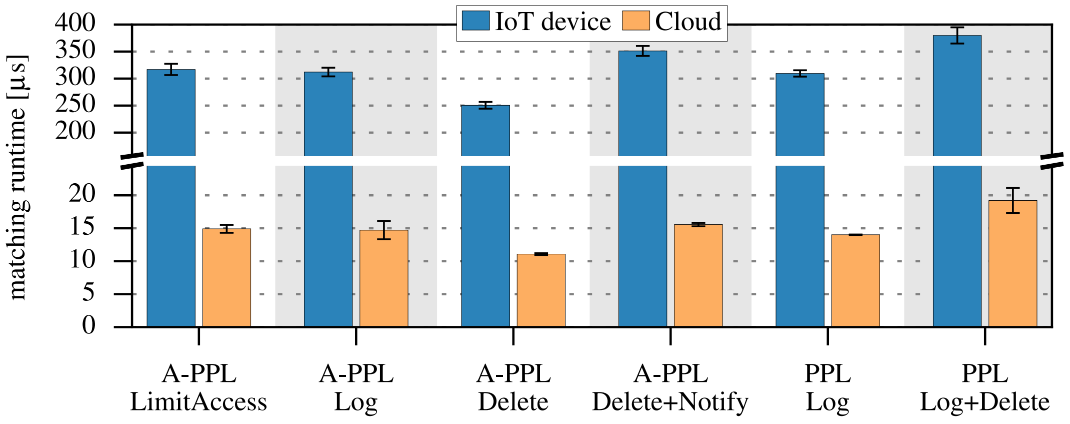

Figure 10: **Real-world matching runtime for IoT and cloud class devices.** A cloud server is able to perform tens of thousands matchings per second. Even on IoT devices, CPPL can perform thousands of matchings per second.

Furthermore, we also evaluated the matching performance of CPPL based on these real world policies. To this end, we employed a cloud-level device but also a typical device from the IoT domain. While cloud systems process user data and therefore often need to perform matching of CPPL policies and system properties, IoT class devices are responsible to handle data closer to the user, e.g., in edge clouds and fog computing. For our evaluation for cloud services, we use an Amazon Web Services EC2 64-bit instance of type m4.large running Ubuntu 14.04. To measure the performance for IoT devices, we utilize a Raspberry Pi (Model B Revision 2.0) with a 700 MHz ARM11 CPU, 512 MB of RAM, and running Raspbian 8.0.

Our results depicted in Figure 10 show that a cloud server is able to perform more than 52 056 policy matchings per second for our largest real-world policy. For smaller policies, this even increases to more than 67 024 matchings per second. Considering that even Dropbox had on average less than 20 000 insert/update requests per second in June 2015 [9], this matching performance of CPPL enables efficient usage of user policies in cloud environments. For IoT devices, the matching rate still ranges from 2 632 up to 3 155 matchings per second. As a single device of this device class serves less users it also has to handle less data (we observed throughput of 149 messages per second [15]) such that this is still a sufficient performance. Thus CPPL enables the efficient use of user policies for edge cloud computing as well as for large centralized cloud datacenters.

3.2.3. Small Overhead for Real World Data and Processing

For the large use cases that we analyze in SSICLOPS, we provide an extensive description of the envisioned use for policy languages in Section 5. Additionally, we performed initial evaluations in a small local testbed to analyze the feasibility of per-data item policies created with CPPL.

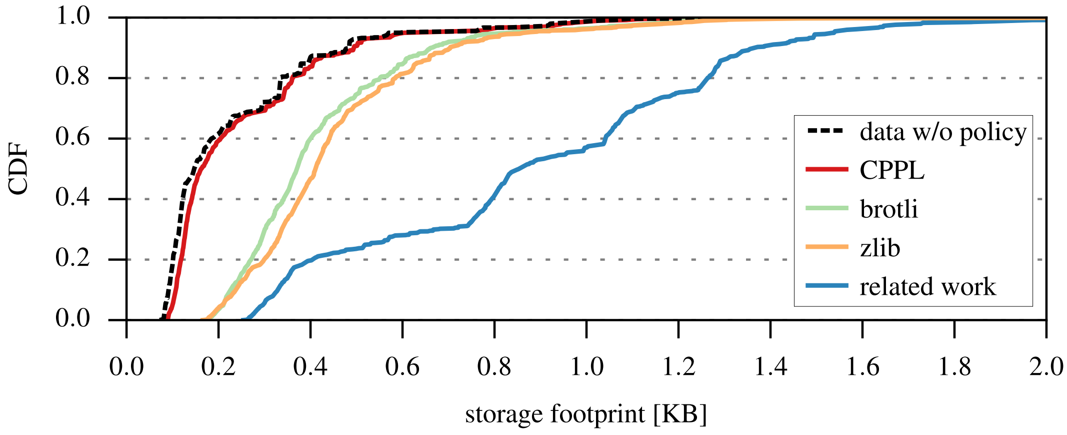

Figure 11: **Impact of policies on storage footprint of real IoT data.** CPPL policies only add a small overhead compared to data without data annotations. Compared to related work and generic compression algorithms, CPPL significantly reduces the storage and communication overhead introduced by the support of user expectations with the help of data annotations.

More specifically, we analyzed the storage overhead introduced by CPPL when using it for IoT data. Furthermore, we evaluated the overhead introduced to machine learning tasks when CPPL is used to control the usage of personal data for such purposes.

To evaluate the impact of per-data item annotations on big data, we take a look at data gathered in the Internet of Things (IoT). In the IoT, data is gathered by a huge number of devices that are typically constrained in their resources and thus often use the cloud for aggregation and processing of gathered data. Thereby, the huge number of IoT devices results in a vast amount of data that needs to be processed. Hence, the upcoming IoT devices increase the amount of transferred data, e.g., up to 40 000 exabytes in 2020 compared to 130 exabytes in 2005 [12]. At the same time, the IoT also significantly increases the diversity of data sources [3], and the granularity of reported data [16]. Thus, analyzing IoT workloads with respect to the storage footprint and communication overhead introduced by CPPL gives us an impression on the impact of CPPL in big data environments.

To study the impact of policy use for IoT data, we sampled frequency and size of real IoT data. To this end, we collected real data of IoT devices from dweet.io [5], a data sharing utility for the IoT. We collected the data over 92 hours and obtained 18.41 million IoT messages gathered by 7 207 distinct devices. The gathered data ranges from 72 byte to 9.73 KB with a mean size of 394 byte. Although they make data publicly available, we protect the privacy of data owners and only stored the identifier of the IoT device, the timestamp of each data message, and the data size, i.e., we did not store the payload of the messages.

In Figure 11, we compare the size of data with and without policies as data annotations. Furthermore, we compare the impact of CPPL policies with that of using alternative policy languages or their compressed forms similarly as in Section 3.2.2. The figure shows the cumulative

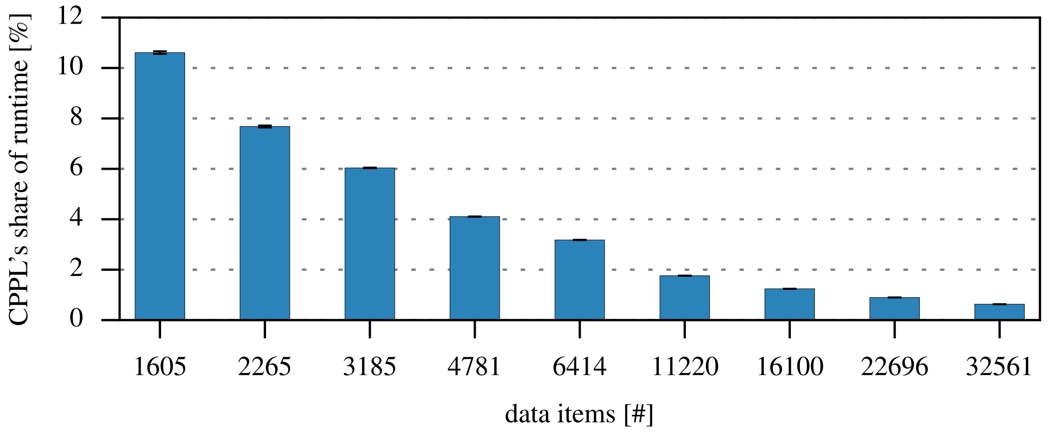

Figure 12: **Impact of CPPL on machine learning (UCI Adult dataset [31]).** For larger data sets, CPPL's share of the runtime becomes negligible.

distribution of IoT message sizes with (solid lines) and without (dashed line) policies attached as data annotations. To attach policies to data items, we uniformly randomly selected one the policies from related work (cf. Section 3.2.2) for each IoT message. The depicted results show that CPPL comes at a negligible storage and transmission overhead compared to plain data that does have policies attached and thus does not support user expectations. Using generic compression algorithms instead of CPPL introduces a significant overhead which further exacerbates when compression is completely neglected. Overall, storing all 18.41 million collected IoT messages without policies requires 4.39 GB. This increases to only 4.68 GB when attaching CPPL policies, 7.86 GB and 8.42 GB for brotli respectively zlib, and a total of 16.37 GB when using policies from related work. Notably, the gathered data only corresponds to less than 4 days of the data created by IoT devices which further increases with the rising number of IoT devices and with longer durations of IoT data storage. Thus, this analysis emphasizes the need for a space efficient privacy policy language mechanism as provides by CPPL with its highly compressed policies with the help of domain specific compression.

With our analysis of CPPL-policy annotated IoT data, we showed that the storage footprint and communication overhead of CPPL is suitable even for big data scenarios. Moreover, we also analyzed the impact of CPPL policy use on the processing, e.g., at cloud systems. To this end, we evaluated the impact of CPPL policies on machine learning which is often used for analysis in big data environments. Machine learning especially profits from the increased amount of information as larger datasets for training the models increases model accuracy [20]. However, the data used for machine learning often contains private data such that per-data item policies can significantly increase the willingness of users to contribute their data to, e.g., medical studies, as users still stay in control for which studies their data is used. Nevertheless, we need to analyze the impact of policy matching on the overall runtime of such studies as policies need to be evaluated before data can be used for training the machine learning model.

To investigate this influence, we measured the runtime of the training phase of the support vector machine LIBSVM [6] when CPPL policies are used to determine if the data item is allowed to be used for this application and compared it to the runtime without such a CPPL policy matching. Again, we uniformly randomly assigned one the policies from related work to each data item. Furthermore, we considered policy initialization overhead for the first occurrence of each domain parameters specification.

Figure 12 shows the fraction of the runtime required for policy matching for different numbers of input records ofs the UCI Adult dataset [32]. The remaining fraction of the runtime is dedicated to the actual training of the support vector machine (SVM). In case of a small number of input records, the processing of policies takes 10.6% of the runtime that is required for the full process (policy matching and training of the SVM). More specifically, policy matching accounts for 18.9 ms while the training of the SVM requires 178.2 ms. However, the fraction of the runtime that is required for policy matching considerably decreases with increasing number of data items as input for SVM training. Considering, e.g., 32 561 data items, policy matching takes only 0.6% (377.7 ms) of the total runtime whereas SVM training is responsible for the other 99.4% (59.8 s). Hence, with respect to the larger datasets that appear in the context of big data scenarios, the runtime overhead of CPPL policy matching is negligible. Concluding, CPPL enables privacy policy-aware machine learning based approaches with almost no overhead on processing time.

4. Hardened security through multipath communications

In the SSICLOPS project, we designed, implemented and tested new secure data transfer protocols for the cloud infrastructure. Our security protocols increase the cost of eavesdropping and tampering with data communications, both between cloud services as well as between a cloud service and Internet clients. Our protocols take advantage of the availability of multiple channels between communicating parties, considering the growing availability of channels between servers, but also between clients and servers (e.g. Wi-Fi, 4G), as well as the growing adoption of multipath TCP [1]. We implemented the protocols as a library that can be transparently loaded by clients and cloud services.

In order to provide secure communications we need two essential methods: a) a key exchange protocol that guarantees security under certain assumptions (in our case this assumption is based on the use of two communication channels); b) a data transfer protocol that uses the key obtained in the previous step to maintain or increase the security level.

Since the proposal of the Diffie-Hellman key exchange protocol [8], many security protocols have tried to provide some sort of guarantee that the exchanged key is secure, i.e. only known by the intended parties. This has been implemented mostly by means of trusted third parties (e.g. certificate authorities, ticket-granting services) or using long-term secrets (passwords and secret keys). One of the most widely used security protocols today for secure communications, Transport Layer Security (TLS), relies on a certificate authority (CA) to authenticate a secret key for use in secure communication, very much in the manner that was proposed by Needham and Schroeder back in 1978 [21].

In our case, we want to provide security for federated clouds, which should be independent of certificate authorities that may very well be untrusted or clearly adversarial. Therefore, we have developed security protocols leveraging the use of multiple public communication channels, striving as much as possible to obtain secure communications even in the presence of the strongest adversaries.

4.1. Secure multipath key exchange (SMKEX)

Our secure multipath key exchange protocol (SMKEX) achieves the best possible security in a multipath scenario, without relaying on long-term secrets or trusted third parties. That is, it can

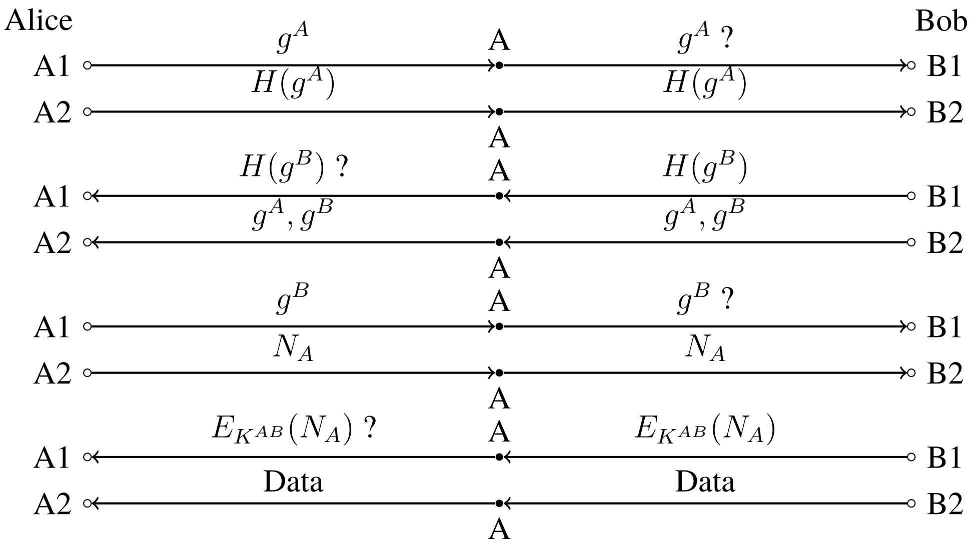

Figure 13: Secure multipath key exchange protocol (*SMKEX*).

resist any combination of attackers as long as there is at least one passive attacker (A-P) or it can resist even active-only attackers, as long as they cannot communicate live/synchronously (A/A). As we show in the next section, our data transfer protocol increases considerably also the cost for adversaries that may even communicate synchronously (A-A).

SMKEX relies on replying the messages received on one path across the other (for now we rely on only two paths). The protocol is shown in Figure 13, where we consider active adversaries that cannot communicate synchronously(A/A). The protocol can be seen as an extension of the Diffie-Hellman key exchange [8] to multiple channels.

One party (Alice) starts by sending her public value g^A (in what follows we omit the modulus for brevity), along with a hash $H(g^A)$ on different channels. The other party (Bob), replies with g^A, along with his public value g^B on the channel where he received the hash value and provides a hash $H(g^B)$ on the channel where he received Alice's public value. In the third step, Alice returns Bob's public value g^B on the other channel than where she received it and sends a fresh nonce N_A on the channel she received g^B. Bob replies with nonce N_A.

Although not shown due to space constraints, we have proved that SMKEX provides a secure key exchange under the stated conditions: at least one passive adversary (A-P) or active adversaries that cannot communicate (A/A).

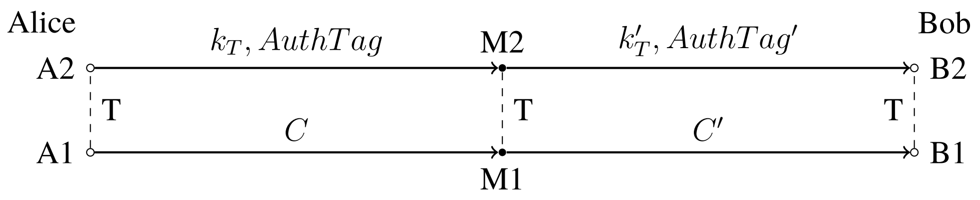

Figure 14: Secure multichannel data transfer protocol (*SMDT*) for the case A-A.

4.2. Secure multipath data transfer (SMDT)

The next step, after having obtained a secure key with SMKEX, is to securely transfer data between the communicating parties. Our main goal is to maintain or even increase the security provided by SMKEX.

A simple approach to maintain the security provided by SMKEX, under the same assumptions (A-P or A/A), is to use authenticated encryption (AE), e.g. AES-GCM, and transfer the encrypted data blocks over the existing channels with an efficient multipath transfer protocol such as MPTCP [1].

However, even for the A-A case (active adversaries that may also communicate synchronously), we may increase the cost of these attackers during data transfer compared to the cost against single-channel opportunistic encryption protocols such as TCPCrypt [4], if we have the following assumptions (in the following we assume a client – Alice – communicates with a server – Bob – over two paths, as in SMKEX):

1. the client and server can force path diversity, i.e. they can force the two paths to be far apart (e.g. across continents). The actual requirement is that the minimum delay T between the two paths be much larger than the maximum delay D between the client and server on each path ($T \gg D$);
2. the client sends N blocks of encrypted data (previously unseen by the attacker) during each transfer to the server. The actual requirement here is that N be sufficiently large to cause some bandwidth overhead to the attackers.[1]

Given the above assumptions, the client can use the protocol from Figure 14 to transfer data encrypted using some authenticated encryption (AE) scheme, such as AES-GCM (we assume that after the SMKEX the client has derived a key K_E for use with AES-GCM).

Say the client originates its data from *A1* and the server receives (i.e. combines) all data at *B1*. In our protocol the client only sends the temporary key k_T (new for each sent message) and the authentication tag over the long-delay (T) channel (*A1-A2-B2-B1*), while the bulk of the encrypted data (C) over the short-delay (D) channel (*A1-B1*). Hence, the total delay from *A1-B1* without an active MITM attack is $2T$ (so the round-trip tip is $4T$) and the total bandwidth used

[1]Examples of such scenarios include uploading videos to Youtube or performing a video conference.

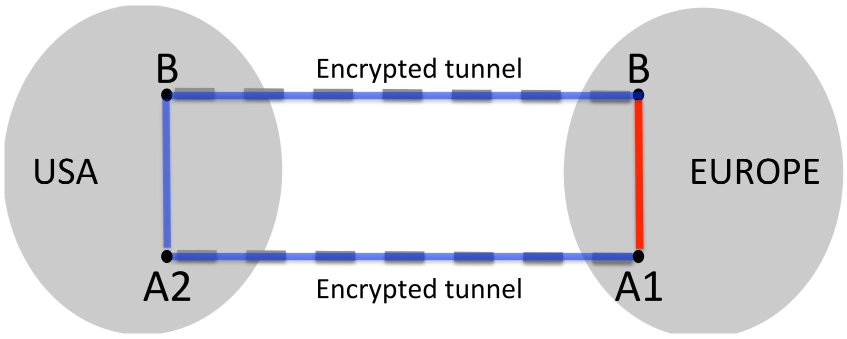

Figure 15: Using long-term tunnels to ensure path and jurisdiction diversity.

across the long-delay channel is only $B = 2$ blocks. An active attacker that wishes to intercept the communication (i.e. decrypt and re-encrypt) needs to perform one of the following: (a) forward all data from *M1* to *M2* as soon as it arrives, so the total delay from *A1* to *B1* is still $2T$. This means the attacker is forced to use a bandwidth of $B = N$ blocks for each message; (b) forward only k_T (necessary for decryption) from *M2* to *M1* and then the new $AuthTag'$ (and optionally k'_T) from *M1* to *M2* (since the adversary needs to re-encrypt). This saves on bandwidth but forces a delay of $4T$ between *A1*-*B1* (so a RTT of $8T$); (c) forge/fake the re-encrypted data C' to avoid the previous costs. Both of the last options may allow an user to detect the attacker, either due to the extra delay or due to the faked data, while the first option incurs a non-negligible cost for the attacker, especially if such attacker performs the attack for all possible connections (which is the more general case).

4.3. Obtaining path diversity

A possible practical option to create path diversity between a client and a server is to use long-lived tunnels that cross jurisdictional and geographical boundaries. To set up such tunnels, Internet users could rely on a trusted cloud server and rent virtual machines in other countries. For example, in Figure 15 a user based in Europe sets up a long-lived tunnel between his machine and its VM in the US. This user will have IP address A1 in Europe, and address A2 inside the USA cloud. The tunnel is secured using a secret decided when the user first registers with the cloud, and will be used to create path diversity for *all connections* this user makes with other parties.[2]

The service provider B must also setup a similar tunnel, but in many cases such tunnels already exist. All major providers, (including Microsoft, Google, Facebook, Amazon, etc.) have multiple datacenters spread around the globe which are already connected via encrypted long-term tunnels. Smaller providers rely on content distribution networks to achieve similar geographical footprint.

[2]This is the only time the user will have to use one of the undesirable solutions such as password-based authenticated key exchange, but this only needs to be done once per user.

On the server side, B must be able to use the connection ID to "recognize" channels belonging to the same client-server connection arriving at different datacenters it owns and direct them to a single backend server. This problem is called multipath load-balancing, but a scalable solution has recently been proposed that we can leverage [22].

Using such solution in combination with SMDT and SMKEX allows us to establish secured multipath communications, that are resilient to most attackers and which can increase considerably the cost of eavesdropping even for the most powerful attackers, all without relying on certificate authorities.

4.4. Implementation

Our protocols run on top of TCP. Both protocol and application data are encapsulated in type-length-value fields which are then exchanged via TCP.

We implemented our protocols in C as a shared library. Our library operates in user space, as a layer between application and the Linux socket API. To simplify deployment, we keep the same function names and signatures. Applications preload the library at runtime, thus ensuring that our functions are called instead of the standard socket API. Small changes are currently required to applications that wish to use our protocols; applications must use our custom `connect2` call to initiate two connections (with different paths) to a server. We intend to make the process fully transparent in the future.

The following functions are currently implemented:

- `socket` - creates a new custom socket (called an *mpsock* socket) capable of sending and receiving information across multiple connections;
- `connect2` - initiates multiple connections to a server address; also runs SMKEX once two valid connections are established;
- `bind` - binds a name to an *mpsock* socket;
- `listen` - starts listening for connections on an *mpsock* socket;
- `accept` - waits for connections from clients; returns once the SMKEX protocol successfully completes with a client;
- `send` - sends application data using SMDT;
- `recv` - receives application data using SMDT.

5. CPPL Scenario Integration

The evaluation of our privacy policy language CPPL already showed its suitable performance for cloud use cases. In this section, we discuss the use of policies in five use cases in the context of the SSICLOPS project. More specifically, we highlight the usefulness of policy support for OpenStack to showcase the policy-aware creation of resources, the in-memory database Hyrise as example for policy use in storage systems, High-Energy Physics workloads as example for big data use cases, and a malware analysis service for insights regarding policy-aware Content Delivery Networks (CDN). Integrating CPPL support into these use cases and evaluating it in the larger interconnected SSICLOPS testbed will give us a more in-depth knowledge of the effect of policy support via data annotations for cloud use cases.

5.1. OpenStack as a Common Foundation

OpenStack[1] is an open source project that provides the tools for hosting cloud services corresponding to the *Infrastructure as a Service* (IaaS) model. A minimal OpenStack installation consists of a compute service for hosting *Virtual Machines* (VMs), as well as services for providing networking and storage infrastructures. Last but not least, an authentication service is required to complete the setup. However, a plethora of additional services is available for extending the functionality of an OpenStack setup [28]. OpenStack enjoys great popularity both in the industry as well as in the academic community [29]. In the context of the SSICLOPS project, all project partners agreed to use OpenStack as the foundation for researching cloud federation strategies due to its relevance, its open source character, and the vivid community.

With OpenStack providing the common foundation for all use case scenarios evaluated within the scope of SSICLOPS, this section documents our efforts of integrating support for *CPPL* within OpenStack. With policy support in place, users of OpenStack gain a fine grained tool for dictating privacy-related terms towards OpenStack instances. In the following sections, we provide an overview of prior policy concepts in OpenStack. Afterwards, we discuss the major design decisions that influenced our implementation strategy. Finally, we demonstrate two examples how support for policy attributes can be easily implemented without having to perform extensive changes on the OpenStack code base.

[1]https://www.openstack.org/

5.1.1. Prior Policy Concepts in OpenStack

Here, we provide a brief overview of approaches for supporting policies in OpenStack that existed prior to our work.

5.1.1.1. oslo.policy

Even though OpenStack is comprised of many distinct projects with strictly separated concerns, certain projects may or sometimes even must use common facilities. To provide a centralized point of contact, the *Oslo* project provides a library for managing inter-project communication. Among others, the *Oslo* project provides unified interfaces for accessing databases and message queues as well as libraries for caching, logging and configuration storage.

Additionally, the *Oslo* project incorporates the package *oslo.policy*, which defines a format for specifying rules and policies and provides a corresponding policy execution engine. However, this policy engine does not suffice the requirements of (federated) clouds, as *oslo.policy* is mainly intended to be used for authorization purposes. Potentially, *oslo.policy* might be used to guard other request properties to which a yes or no answer would be sufficient, i.e., is the user allowed to instantiate a *Virtual Machine* in the United States of America. However, in the context of *SSICLOPS*, more complex policies ought to be supported [10, 11].

5.1.1.2. Swift Storage Policies

Swift is the OpenStack service for object storage. Policy support is provided at the fundamental level, as the containers holding objects can be annotated with policies such as replication rate. Policies are defined by users during the creation of a container, however policies are restricted to core concerns of the object storage and may not be used by other OpenStack services.

5.1.1.3. Policy-based Scheduling in Nova

Upon creation of a new virtual machine, OpenStack has to decide on which host it should be instantiated. The Nova scheduler therefore attempts to locate a host that fulfills several predefined policies, including: [27]

- A sufficient amount of main memory must be available to start the VM.
- In case a specific hypervisor has been requested, it must be available on the target host.
- Only hosts that belong to the user-specified availability zone may be used to host a VM.
- The quota of the user must be considered.

Using this policy-based scheduling approach employed by the Nova scheduler enables restricting VM instantiation requests to certain hosts. [19] Custom policies can be added by OpenStack instance operators by extending the policy set employed by the Nova scheduler. Using this mechanism, it would be easy to implement a policy that makes sure that VMs of specified users are transparently instantiated in a certain region only.

The major disadvantage of this approach is that users can neither review nor edit the policies that are applied to requests. Only OpenStack operators are able to add, review or edit policies. Another shortcoming of this approach is that the policy support is restricted to the Nova component.

5.1.1.4. Group Based Policies in Neutron

The OpenStack networking component *Neutron* employs the concept of *Group Based Policies*. In the context of networking, policies can be used to specify the treatment of packets based on certain properties (e.g. the employed protocol or port). However, due to performance reasons, policies are transformed into virtual networks (including switches, router and firewalls) that satisfy the requirements rather than using a per-packet enforcement level. As this concept is very specific to the networking use case, it cannot be re-used in other OpenStack components in order to facilitate policy support.

5.1.1.5. Congress

Swift Storage Policies, Policy-based Scheduling and Group Based Policies are all internal mechanisms for various OpenStack components for evaluating policies. However, all approaches have in common that they cannot be used by other components and that they are specifically tailored to the respective domains. The *Congress*-project is a dedicated OpenStack service that aims at providing a centralized policy component for enabling compliance in cloud-based environments. As a consequence, all preceding approaches for supporting policies in OpenStack could be implemented using *Congress*. [30, 31]

Congress uses a monitoring approach in order to maintain a high degree of independence among OpenStack services. It detects policy violations in a passive mode of operation by querying the state of all involved OpenStack services in regular intervals using their corresponding APIs. In case the state of an OpenStack service deviates from a policy, the violation is logged and notifications can be triggered if configured

One major limitation of *Congress* is that its monitoring-based approach impedes the implementation of actual enforcement mechanisms. Policies may specify how violations should be treated. In addition to logging violations and triggering notifications, policies can also be configured to revert policy violations. However, several actions are hard to revert, especially in cases where the violating action triggers many side-effects that have to be reverted as well.

Policies in *Congress* are expressed using the declarative *Datalog* policy language, which is comprised of a subset of the *Prolog* programming language. The *Congress* API is intended to be used by OpenStack instance operator. However, using *oslo.policy*, the API can be made available for users.

In order to quantify the maturity of OpenStack projects, the OpenStack Foundation employs a maturity scale ranging from 1 to 8. After 2 years of development, *Congress* earned the lowest score 1 [24] for the Mitaka release (April 2016).

5.1.2. Design Decisions

In order to provide a better understanding of the design we came up with for our policy integration approach in OpenStack, we provide a brief discussion of some of the most crucial aspects that strongly influenced the design of our approach.

5.1.2.1. Monitoring versus Proactive Adherence

As elaborated in the context of *Congress* (see Section 5.1.1.5), proactively adherence to policies requires numerous changes in the OpenStack code base compared to a monitoring-based approach. However, the proactive approach never allows policy violations to occur in the first place, whereas monitoring-based approaches are limited to reacting on policy violations by means of logging and issuing counteracting actions. Here, we decided to aim for the proactive approach.

5.1.2.2. Versatility of Policies

The SSICLOPS policy language *CPPL* (see Chapter 3) supports a wide range of policy attributes. Due to this versatility of *CPPL*, policies might affect an arbitrary amount of OpenStack services. In order to deal with this high degree of versatility, each OpenStack service had to be adapted in order to support *CPPL*.

5.1.2.3. Development Process of OpenStack

OpenStack services are strictly separated in order to prevent inter-service dependencies. This level of isolation is also reflected by the development process: Services may only interact using their regular, public APIs and twice per year during the OpenStack Summits, developers meet to discuss and plan the implementation efforts 6 months ahead. [25] Contributing code to OpenStack projects involves a very complex workflow, which goes far beyond the usual habits of the typical fork-pull workflow applied on many GitHub projects. Die OpenStack sources on GitHub are merely a mirror, whereas the actual sources are maintained on an OpenStack-specific Git-server[2], which held roughly 1600 repositories by November 2016.

To contribute code to OpenStack, the following process has to be adhered to: [23, 26]

- You need to have an account on the platform https://launchpad.net/.
- You need to be a member of the OpenStack Foundation.
- You have to accept the license agreement.
- The feature to be implemented has to be discussed based on a specification. Several projects are using dedicated repositories for keeping track of specifications.
- A so-called *Blueprint* is created on *Launchpad*, pointing to the specification of the feature. The *Blueprint* is used to track the progress of the feature.
- The feature is implemented in a dedicated branch.
- All code is tested extensively before it is adopted in the main branch. For the purpose of testing, the reviewing system *Gerrit*[3] is used. All changes have to be tested using automated tests. Furthermore, the changes have to be accepted manually by human.

As this process introduces a fair amount of complexity, it would not be feasible to perform changes on the many OpenStack projects, as it would be required even by simple policies. Therefore, a central requirement for us was to keep the overhead for implementing policies as low as possible. This decision strongly influenced the design of our *policyextension* framework, which is outlined in Section 5.1.3.

5.1.3. Integrating Policy Evaluation into OpenStack Components

One of the fundamental hurdles towards integrating proactive policy evaluation into OpenStack is the tremendous implementation effort, as all services affected by a policy have to be altered significantly.

Our implementation strategy, the *policyextension*-framework aims at providing the following characteristics:

- Interpretation and evaluation of policies should be implemented in one single location, rather than being spread across the code base of numerous OpenStack services.
- Integrating policy support should be minimally invasive regarding code changes in existing services.
- Easy maintainability of policy support code.

[2]https://git.openstack.org/cgit/
[3]https://review.openstack.org/

- Facilities should be easily extensible in order to support additional policy attributes.

To realize these goals, our *policyextension*-framework uses *PolicyExtensions* in order to integrate the evaluation of policy attributes. *PolicyExtensions* share many characteristics with plug-ins, as they are not part of the original code base. In contrast to plug-ins however, *PolicyExtensions* do not rely on plug-in mechanisms but inject their code at the locations of their own choice. The infrastructure for injecting *PolicyExtensions* is provided by the *policyextension*-framework, which also defines a certain format that *PolicyExtensions* have to adhere to by inheriting from a specific base class.

In this document, we are going to skip over the implementation details of the *policyextension*-framework itself, however we demonstrate its capabilities by presenting two examples for valid *PolicyExtensions*.

5.1.3.1. Policy Example 1: Disk Encryption

To evaluate a policy, it has to be interpreted first. The *policyextension*-framework uses the dictionary data type `dict` in *Python* to express policies in the form of key-value pairs. Our implementation of CPPL can be instructed to output such a list as result of the matching process. The following listing shows an example output for a policy that formulates the use of disk encryption:

```
{
 "storage": {
  "encryption": True
 }
}
```

To support such a policy, we adapted the *Cinder* service of OpenStack. In OpenStack, the Cinder service is responsible for providing *Block Storage*, which is indicated by the `storage` key. The embedded key `encryption` with the corresponding boolean value true then specifies, that newly created volumes must use encryption. Below, the example code demonstrates how the proactive policy adherence can be implemented by creating a new *PolicyExtension*:

```
from policyextension import PolicyExtensionBase, PolicyViolation
from cinder.volume import volume_types

class CinderEncryptedVolumeTypeRequiredExtension(PolicyExtensionBase):
  func_paths = ['cinder.volume.api.API.create']

  def create(self, func_args, policy):
    try:
      if policy['storage']['encryption']:
        volume_type = func_args['volume_type'] or
            volume_types.get_default_volume_type()
        if not volume_type or not
            volume_types.is_encrypted(func_args['context'],
            volume_type['id']):
            msg = "Your␣policy␣requires␣using␣an␣encrypted␣volume␣type."
            raise PolicyViolation(msg)
    except KeyError:
      pass
```

5.1.3.2. Policy Example 2: Restriction on Availability Zones

As a second example, we demonstrate the code for supporting a policy that restricts the instantiation of VMs to a set of whitelisted availability zones. Here, we are using the OpenStack mechanism of availability zones in order to model geographic locations:

```
from policyextension import PolicyExtensionBase, PolicyViolation
import random

class AvailabilityZoneRestrictionExtension(PolicyExtensionBase):
    func_paths = ['nova.compute.api.API.create']

    def create(self, func_args, policy):
        availability_zone = func_args['availability_zone']
        try:
            az_whitelist = policy['availability_zones']
            if availability_zone:
                if availability_zone not in az_whitelist:
                    msg = ("Your␣policy␣does␣not␣allow␣the␣availability␣zone␣you␣
                        selected.")
                    raise PolicyViolation(msg)
            elif az_whitelist:
                func_args['availability_zone'] = random.choice(az_whitelist)
        except KeyError:
            pass
```

With these examples, we conclude the presentation our approach for integrating proactive policy support within OpenStack. The core contribution of the *policyextension*-framework is that it grants the infrastructure for implementing support for various policy attributes with minimal effort and in a centralized component, even in cases where supporting policy attributes may involve multiple OpenStack services.

5.2. In-Memory Databases in the Cloud

Hyrise-R(epilication) is a scale-out extension for the in-memory research database Hyrise. A Hyrise-R cluster consists of a query dispatcher, a single Hyrise master instance, and an arbitrary number of replicas. Users submit their database requests to a query dispatcher, which acts as a load balancer for reading queries. The dispatcher parses the queries for data-altering operations, i.e., inserts, updates, and deletes. The master node processes all writing transactions. Data changes are written into a local log. Besides storing the log entries to persistent memory, the master sends them to the replicas, which update their data accordingly.

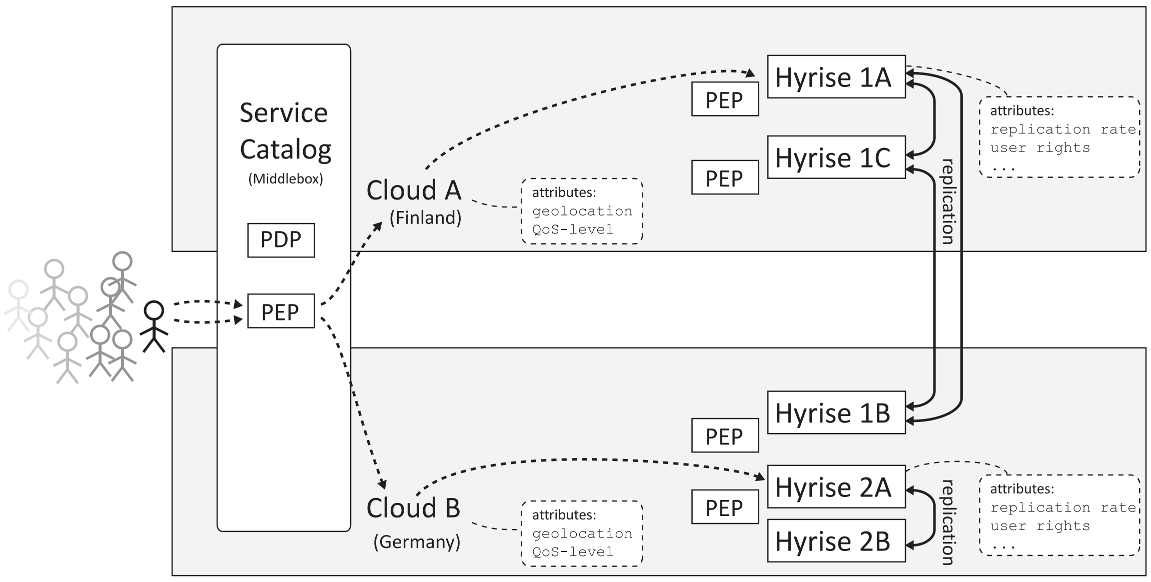

Figure 16: Use case scenario: Users request instances of the *Hyrise-R* in-memory database and annotate their requests with certain policy demands. The *policy decision point* (PDP) acts as the initial entry point and routes requests through a series of *policy enforcement points* (PEP) to process the requests accordingly.

Figure 16 shows two Hyrise-R clusters in a cloud environment. Cluster *one* comprises the Hyrise instances 1A, 1B, 1C. The second Hyrise-R cluster consists of the Hyrise instances 2A and 2B. The dispatchers are not illustrated and may be deployed in- or outside of the cloud environment. The Hyrise instances 1A and 2A act as masters. A number of policy attributes can be used to describe database systems, and as those the two Hyrise-R clusters. Many policy attributes are common for commercial database system. In the following we describe the most relevant and how they are or can be implemented in Hyrise-R.

Replication Factor: The replication factor describes the number of Hyrise instances in a cluster. Each Hyrise instance consists of processing units for query processing and a database log as durable storage. Using shared logs or multiple logs per instance may result in different replication rates for in-memory data structures and the persistency layer. The replication factor and replication models influence the availability.

Geolocation: The geolocation describes the geographical location of cluster nodes on which the Hyrise instances run. Legal requirements may impose or forbid storing or processing data at specific locations. Besides a geolocation of a server close to the database client may reduce the query latency.

Access Control: Access control is the selective restriction of access to data. In order to enable effective access control mechanisms, user account management is necessary. Access control is an important and common feature of commercial database system. Access control can be implemented and enforced inside Hyrise instances or the dispatcher.

Notification: A notification system triggers actions as result of event, e.g., writing an email as reaction of a data tuple modification. Similar to access control, notification systems are common for commercial database systems and can be added to Hyrise.

Storage Type: Database structures can be stored on different storage types, differing in access speed, capacity, durability, and costs. In-memory databases usually store all database table data in main memory. However, infrequently accessed portions of the data may be stored on slower storage types with larger capacities to save costs and deploy bigger systems. Besides, for logging, one can use SSDs or Non-Volatile RAM.

Security: A couple of policy attributes describe the security of a database system, e.g., the encryption of table objects and the tenancy of the database instance in the cloud.

5.3. High-Energy Physics Workload

In the High Energy Physics (HEP) scenario the focus is on storing data on high-energy particle collisions produced by the four Large Hadron Collider (LHC) experiments at CERN: ATLAS, CMS, LHCb, and ALICE. This event data is handled by a distributed computing infrastructure called the Worldwide LHC Computing Grid (WLCG). The idea of WLCG is to identify and authenticate the user and provide him/her with a transparent access to the requested data and computing resources. The user does not typically need to know the actual location of data and computing resources.

The required storage and computing resources for the LHC data analysis are distributed to hundreds of data centers on three levels called tiers: tier-0, tier-1 and tier-2. The tier-0 datacenter is located in two CERN sites (Geneva, Switzerland and Budapest, Hungary). These datacenters store the first long-term copy of the raw physics data. This data is replicated to 13 tier-1 sites and further to around 160 tier-2 sites. Tier-1 sites are typically large computer centers and tier-2 sites scientific institutes such as universities. In addition, any other sites that host the LHC data might be sometimes called tier-3 sites, but they don't have a formal WLCG role.

Typically tier-2 sites contain only the data that is interesting for the researcher working in the institute and not the full copy of all LHC data. If a researcher tries to access data that is not available on his/her own site, it is either copied from a tier-1 site or alternatively from another tier-2 site. The tier sites are free to select the storage technology they want to use, such Ceph or dCache, to provide the storage space they are required to have available. The same applies to the storage protocols, which vary from standard protocols such, as HTTP, to HEP specific protocols, such as XRootD.

The total WLCG storage capacity is currently around 300 Petabytes. The produced HEP event data is stored on database like containers, called ROOT files, which are typically from few hundred megabytes to few gigabytes. ROOT files store data in a tree-like data structure that is optimized for accessing fast huge amounts of event data. The ROOT files are in a compressed binary format and they store both the data and description of the data. The ROOT files can also contain summary information, such as histograms. One of the main features of HEP computing is that typically only a part of each ROOT file is read during processing. This means that both storage system and the data access protocol need to support byte range requests, also known as vector reads.

The examined HEP scenario focuses on XRootD, which is name of both a storage system and a data access protocol. XRootD is specifically designed for HEP data requirements and it is also the basic building block for the disk-based storage software, called EOS, used in the tier-0 site at CERN. The XRootD protocol is widely available on HEP software packages such as ROOT. The XRootD server has a plug-in architecture that allows translation between site-specific protocols, such as dCap, and XRootD protocol. For this reason, it can be used to interface a wide range of storage systems used in the WLCG.

One of the main features of XRootD is that it can redirect traffic to the correct location, if the request data is not found from the local storage server. The data is then transferred directly to the client from the storage system that is actually storing the data. The authorization and authentication, which is usually required to access the data, are also handled by the XRootD server. Usually the data access is authorized to all members of the experiment's collaboration, such as the CMS experiment collaboration. The same data should always have the same access rights, regardless where it is actually stored. It is usually up to the tier sites to implement the correct access.

The data sets have different replication policies based on the tier sites. Most commonly Raid-1 is used, i.e. two separate copies of data is stored. Higher replication factors are used for the most popular data sets. Exact replication policies differ between experiments and are adapted as needed. For example replication policy of ATLAS experiment is reviewed at least yearly based on operational experience and storage space constraints. Automated popularity based replication methods are also used.

Deletion policies evolve and vary between experiments also like replication policies. Especially the number of copies is varied based on the data relevance and popularity for physics analysis. A dedicated Site Cleaning Agent is utilized to analyze what data is used and it is able to propose data that can be safely deleted. In addition to this, manual data deletion campaigns are regularly organized to increase the available disk space. In some sites special "data lifetime model" is used to specify when data is to be removed or moved from a disk to tape. Applying of the "data lifetime model" is not automated but rather manual process.

The LHC experiments rely on Data Popularity Service to keep track what was used, how much it was used, who used it and where it was used. The service operates by keeping track of opened files. A set of dedicated plugins is used, e.g. in XRootD, to collect this data and store in a Popularity database. This service is important both for data replication and deletion.

In summary there is several data policies governing HEP data in WLCG. Most of the time the data policies are rather dynamic and the actual applying of these policies is left for a group of separately and differently operated sites. More formal data policy language could improve the reliability and efficiency of storage space utilisation.

5.4. Content Distribution & Caching

F-Secure is providing various security products that are consuming intelligence and data from back-end services and producing data for these services. High level service types are licensing and product usage, anonymous telemetry and reputation delivery and security analysis. Different service classes are segregated and there is no linkage between different systems, so that systems personal data can be kept minimum. For example the reputation delivery and security analysis systems do not have a linkage to identity of the end user.

All of these products have privacy policies that dictate on which data is collected by the services, where the data can be stored, and how it can be used. Currently there is no formal way to define how that product privacy policy is applied in an uniform way. Instead privacy policy is something that translates to product features, product options, back-end configuration options, and back-end deployments. Many of these have different stakeholders or implementation teams.

We envision that by utilizing the privacy policy language it would be possible to be more formal in defining functionality and configuration and also bridge the gap between legal, product management, privacy savvy end customers, and finally implementation. With this approach it would be possible to build common core services that adhere to policies transmitted with the data. This would also transparently bring policy attributes to lower level services without modifying the path/micro-services that do not need to know certain piece of the policy. For example adding an attribute whether some data item can be used for machine learning training can only be added on the sender side and only the service that will check that attribute might be the training process and all the other services can just ignore that property.

As one specific scenario, we describe the benefits of policies for the F-Secure Security Cloud that has a file and URL reputation service that provides security, classification, and prevalence information based on file or URL hashes (e.g., used to detect malicious user supplied data). To obtain the reputation of a file or an URL, the users sends its hash to the Security Cloud which returns the reputation stored for the obtained hash. However, when the data was never seen before a hash cannot help to give a reputation. In this case content itself can be sent to the F-Secure Security Cloud for further analysis. Policies can, for example, set out the type of analysis that the end customer permitted to be carried out (automatic static and/or runtime evaluation vs. analysis by humans), where to store the data and when to delete it. Same or derived policies may also apply to extracted and derived metadata of the original data item.

In the following, we point out policy attributes that are beneficial for the F-Secre Security Cloud. Initially, data has to be collected to create a reputation based on analysis. Policies can control *when it is allowed to collect* such data. Furthermore, users and customers may require the cloud to *notify* them about actions such as storage of data.

Also, restrictions regarding the processing of data are required by customers. Policies can specify the allowed *depth of analysis* which may range from automated security analysis to security analysis by a human being. Additionally, policies allow for specification of *access restrictions* on data and for restrictions regarding the *location* of processing. Furthermore, data classifiers often employ machine learning which use input data for training its model. A policy allows customers to allow or deny the use for this *machine learning training*. Similarly, policies can control the *confidentiality level of data after analysis*, e.g., if malicious data should be kept confidential or become public. Finally, policies can *specify the allowed level of data aggregation*.

While and after processing, the cloud has to store the data. Here, customers can leverage policies to exclude certain *cloud storage providers*, e.g., competitors, specify the required *security level* such as encryption, and restrict the *location* of the data store. Similarly, and also affecting data processing, a policy can further list suitable security architectures for storage and compute systems, i.e., customers may require the use of trusted computing support and even list specific types of this technology. Finally, policies enable customers to specify a *time or event-based deletion of data*, e.g., immediately after analysis or after a certain time.

6. Conclusions

In this deliverable, we outlined the use of privacy policy languages and the design of new multipath security protocols within federated clouds, i.e., clouds that comprise of multiple smaller clouds each handled by distinct cloud providers.

Furthermore, we presented an extensive evaluation of our compact privacy policy language (CPPL) that we designed especially considering the requirements of federated cloud computing. This evaluation reveals significant improvements over existing work. CPPL considerably decreases the storage footprint of policies enabling a feasible use of per-data item policies in cloud and federated cloud scenarios where data needs is often transmitted, e.g., from storage to processing systems. Moreover, the underlying compression efficiently scales with the size of the policy and the comprehensiveness of the domain parameters. Although it introduces some still small runtime overhead, this is limited as compression of policies is required only once. Furthermore, the efficient matching of CPPL policies with system properties adds limited overhead to the processing of data, e.g., in the case of training of machine learning models. Thereby, the matching runtime is mostly determined by the policy size such that even very comprehensive domain parameters that enable a high variety in the used policies can be used without significantly increasing this runtime. In conclusion, CPPL enables efficient transmission, storage, and evaluation of policies enabling cloud providers to respect user expectations.

Having shown the feasibility of per-data item annotations for cloud computing, we analyzed the benefits of policy support in our SSICLOPS use cases. Policy support for OpenStack enables the policy-aware creation of virtual machines in a (federated) cloud. This, e.g., can serve as a basis to bootstrap new resources that process data that could not be handled by the cloud beforehand because of missing application instances that comply with (often) requested user expectations. In-memory databases like Hyrise profit from policy support as it allows to respect the expectations of users regarding storage of their data. Private users and enterprises require such support, e.g., to adhere to legal rules that otherwise would make them unable to use the cloud for storage of corresponding data. We identified a wide range of useful policy attributes for cloud storage services ranging from location of the data centers, access restrictions, and storage type to user notification on actions like modification or even specifying backup or replication strategies within the policy. In big data scenarios like our High-Energy Physics use case automated analysis and adherence to policies significantly helps to increase the reliability and efficiency of storage space utilization. Data analysis services, e.g., checking for malicious user data supplied to web services, profit from the use of privacy policies as they allow an in depth specification of the allowed handling of user supplied data. For example, when analyzing data for malicious content, this data may not be allowed to be reviewed by a human due to privacy reasons. Overall, the different use cases show the broad scope of policies in (federated) cloud scenarios and how they can help to improve these services in many ways.

Bibliography

[1] A. Ford and C. Raiciu and M. Handley and O. Bonaventure. *RFC6824:TCP Extensions for Multipath Operation ...* https://tools.ietf.org/html/rfc6824.

[2] M. Azraoui, K. Elkhiyaoui, M. Önen, K. Bernsmed, A. S. Oliveira, and J. Sendor. "A-PPL: An accountability policy language". *DPM*. 2014.

[3] P. Barnaghi, W. Wang, C. Henson, and K. Taylor. "Semantics for the Internet of Things: Early progress and back to the future". *IJSWIS* (2012).

[4] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. "The Case for Ubiquitous Transport-Level Encryption." *USENIX Security Symposium*. 2010, pp. 403–418.

[5] Bug Labs, Inc. *dweet.io – Share your thing like it ain't no thang*. https://dweet.io/.

[6] C.-C. Chang and C.-J. Lin. "LIBSVM: A library for support vector machines". *ACM TIST* (2011).

[7] R.-A. Cherrueau, R. Douence, H. Grall, J.-C. Royer, M. Sellami, M. Südholt, M. Azraoui, K. Elhhiyaoui, R. Molva, M. Önen, A. Garaga, A. S. Oliveira, J. Sendor, and K. Bernsmed. *Policy representation framework*. Tech. Report. A4Cloud Consortium, 2013.

[8] W. Diffie and M. Hellman. "New directions in cryptography". *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654.

[9] Dropbox Inc. *400 million strong*. 2015. URL: https://blogs.dropbox.com/dropbox/2015/06/400-million-users/.

[10] F. Eberhardt, J. Hiller, S. Klauck, M. Plauth, A. Polze, and K. Wehrle. *D2.2: Design of Inter-Cloud Security Policies, Architecture, and Annotations for Data Storage*. Tech. rep. Jan. 2016.

[11] F. Eberhardt, M. Plauth, A. Polze, S. Klauck, M. Uflacker, J. Hiller, O. Hohlfeld, and K. Wehrle. *D2.1: Report on Body of Knowledge in Secure Cloud Data Storage*. Tech. rep. June 2015.

[12] J. Gantz and D. Reinsel. *The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east*. IDC iView. 2012.

[13] M. Henze, M. Grossfengels, M. Koprowski, and K. Wehrle. "Towards Data Handling Requirements-Aware Cloud Computing". *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. Vol. 2. Dec. 2013, pp. 266–269.

[14] M. Henze, R. Hummen, and K. Wehrle. "The Cloud Needs Cross-Layer Data Handling Annotations". *Security and Privacy Workshops (SPW), 2013 IEEE*. May 2013, pp. 18–22.

[15] M. Henze, J. Hiller, S. Schmerling, J. H. Ziegeldorf, and K. Wehrle. "CPPL: Compact Privacy Policy Language". *Proc. 2016 ACM on Workshop on Privacy in the Electronic Society*. WPES '16. ACM, 2016, pp. 99–110. DOI: 10.1145/2994620.2994627.

[16] R. Hummen, M. Henze, D. Catrein, and K. Wehrle. "A Cloud Design for User-controlled Storage and Processing of Sensor Data". *CloudCom*. 2012.

[17] Intel IT Center. *Peer Research: What's Holding Back the Cloud?* Tech. rep. 2012.

[18] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun. "Home is Safer Than the Cloud!: Privacy Concerns for Consumer Cloud Storage". *Proc. Seventh Symposium on Usable Privacy and Security*. SOUPS '11. ACM, 2011, 13:1–13:20. DOI: 10.1145/2078827.2078845.

[19] Khanh-Toan Tran and Jérôme Gallard. *A new mechanism for nova-scheduler: Policy-based Scheduling*. 2013. URL: https://docs.google.com/document/d/1gr4Pb1ErXymxN9QXR4G_jVjLqNOg2ij9oA0JrLwMVRA/edit.

[20] S. Lohr. "The age of big data". *New York Times* (2012).

[21] R. M. Needham and M. D. Schroeder. "Using encryption for authentication in large networks of computers". *Communications of the ACM* 21.12 (1978), pp. 993–999.

[22] V. Olteanu and C. Raiciu. "Datacenter-scale Load Balancing for Multipath Transport". *Proc. 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox '16. 2016.

[23] OpenStack Foundation. *Blueprints*. Nov. 2016. URL: https://wiki.openstack.org/wiki/Blueprints.

[24] OpenStack Foundation. *Congress*. Nov. 2016. URL: https://www.openstack.org/software/releases/mitaka/components/congress.

[25] OpenStack Foundation. *Design Summit*. Nov. 2016. URL: https://wiki.openstack.org/wiki/Design_Summit.

[26] OpenStack Foundation. *Developer's Guide*. Nov. 2016. URL: http://docs.openstack.org/infra/manual/developers.html.

[27] OpenStack Foundation. *Filter Scheduler*. Nov. 2016. URL: http://docs.openstack.org/developer/nova/filter_scheduler.html.

[28] OpenStack Foundation. *OpenStack Services*. Nov. 2016. URL: https://www.openstack.org/software/project-navigator.

[29] OpenStack Foundation. *OpenStack User Stories*. Nov. 2016. URL: https://www.openstack.org/user-stories/.

[30] OpenStack Foundation. *Policy as a service ("Congress")*. Nov. 2016. URL: https://wiki.openstack.org/wiki/Congress.

[31] OpenStack Foundation. *Welcome to Congress!* Nov. 2016. URL: http://docs.openstack.org/developer/congress/.

[32] J. C. Platt. "Fast training of support vector machines using sequential minimal optimization". *Advances in Kernel Methods*. 1999. Chap. 12.

[33] *PPL FI-WARE data handling generic enabler*. GitHub, https://github.com/fdicerbo/fiware-ppl.