CPPL: Compact Privacy Policy Language

Martin Henze, Jens Hiller, Sascha Schmerling, Jan Henrik Ziegeldorf, Klaus Wehrle
Communication and Distributed Systems, RWTH Aachen University, Germany
{henze, hiller, schmerling, ziegeldorf, wehrle}@comsys.rwth-aachen.de

ABSTRACT

Recent technology shifts such as cloud computing, the Internet of Things, and big data lead to a significant transfer of sensitive data out of trusted edge networks. To counter resulting privacy concerns, we must ensure that this sensitive data is not inadvertently forwarded to third-parties, used for unintended purposes, or handled and stored in violation of legal requirements. Related work proposes to solve this challenge by annotating data with privacy policies before data leaves the control sphere of its owner. However, we find that existing privacy policy languages are either not flexible enough or require excessive processing, storage, or bandwidth resources which prevents their widespread deployment. To fill this gap, we propose CPPL, a Compact Privacy Policy Language which compresses privacy policies by taking advantage of flexibly specifiable domain knowledge. Our evaluation shows that CPPL reduces policy sizes by two orders of magnitude compared to related work and can check several thousand of policies per second. This allows for individual per-data item policies in the context of cloud computing, the Internet of Things, and big data.

1. INTRODUCTION

Cloud computing, the Internet of Things (IoT), and big data lead to an increasingly interconnected world, where new data sources continuously emerge [16, 21]. This is accompanied with a massive growth in the amount of data, fundamentally changing data processing: the prevalence of local processing is superseded by processing data outside of the territory of data owners. Besides enormous benefits such as availability, scalability, and cost efficiency, we also face severe privacy challenges [16,17,29,35,38]: Sensitive data that is transferred out of trusted networks might be inadvertently forwarded to third-parties, used for unintended purposes, or handled violating legal requirements. These privacy concerns, missing trust, and legal restrictions on data handling prevent a wide range of users and companies to fully embrace the advantages of an interconnected world [18, 29].

WPES'16, October 24, 2016, Vienna, Austria

ISBN 978-1-4503-4569-9/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2994620.2994627

Current state of the art, i.e., legal text dictating privacy policies by providers, can no longer sufficiently address these concerns as the massive growth in the amount of data is accompanied by a significant increase of diversity of data sources [3] and high granularity of reported data [21]. To account for this development, related work proposes to attach *per-data item privacy policies* (also referred to as *sticky policies*) to data before it leaves trusted networks [15,18,28,30,34]. Instead of having a provider dictate a privacy policy for all users, per-data item policies enable each user to specify her own privacy requirements which then have to be enforced by the provider. Such policies enable the user to express her individual privacy requirements down to the level of specific data items. For example, readings of personal medical devices should be treated differently from much less sensitive readings of personal weather stations. This combination of user-centricity and granularity empowers users to effectively remain in control over their data, even if it leaves their physical control.

To realize such fine-granular user-centric policies, related work introduced a wide range of policy languages, either generic or specifically tailored for a specific scenario, e.g., in the area of accounting, banking, handling of insurance information, or processing of medical data of patients. These range from policies for realizing access control [11, 13] over complete data handling policies [1,36] to digital rights management [20]. However, we identify two severe drawbacks of these languages when applying them in an interconnected world: (i) they are either limited in scope and do not provide the necessary expressiveness and flexibility required by increasingly dynamic scenarios and evolving perceptions of privacy, or (ii) they consume excessive processing, storage, or bandwidth resources which becomes prohibitive when considering the frequent exchange of rather small data, e.g., in the context of the IoT. This issue further exacerbates with recent proposals of attaching policies to individual network packets to realize policy-based routing [25].

To overcome these shortcomings and hence offer support for fine-granular user-centric policies in an interconnected world, we propose to introduce a *domain specific* compression step before sending a policy over the network. To this end, we incorporate flexibly specifiable domain knowledge to realize an efficient bit-level compression. More specifically, our contributions in this paper are:

1. We analyze the deployment and network scenarios in an increasingly interconnected world as well as the suitability of privacy policy languages proposed by related work to address emerging requirements in these scenarios. Based

on our analysis, we find a mismatch between the communication patterns in such networks and the characteristics of existing privacy policy languages.

2. We present CPPL, a **C**ompact **P**rivacy **P**olicy **L**anguage designed for the dynamic and high-frequent characteristics of increasingly interconnected networks. CPPL compresses a textual policy specification and an interchangeable domain specification allows us to adapt the domain specific compression to any (even as yet unknown future) deployment and network scenarios.
3. To illustrate the feasibility of CPPL, we first perform synthetic benchmarks and then compare CPPL with other privacy policy languages. Furthermore, we showcase the applicability of CPPL in the context of cloud computing, the IoT, and big data. Our results show that CPPL is able to reduce policy sizes by two orders of magnitude compared to related work and process several thousand of policies per second in real-world settings.

2. PRIVACY POLICIES IN AN INTERCONNECTED WORLD

On our path to an interconnected world, users' personal spheres are inevitably penetrated and support for fine-granular user-centric privacy policy languages becomes a crucial challenge. In this section, we outline our targeted scenario and derive requirements that we argue must be addressed by any viable solution. We then rigorously analyze existing policy languages with respect to these requirements and identify different short-comings that render existing work inapplicable in this scenario.

2.1 Scenario

In this paper, we consider scenarios where data is transferred out of the data owner's control sphere to back-end infrastructures as shown in Figure 1. This scenario is commonplace in cloud computing already today, but even more in the IoT vision of an interconnected world. An IoT home automation system, e.g., might transfer raw sensor data to a cloud back-end to infer a user's presence and activity for optimal control of HVAC appliances. With the upcoming trend of big data, masses of data will be used to derive novel insights. These scenarios all have in common that, while considering huge amounts of data in total, individual data pieces are comparably small. For example, single IoT measurements can be as small as 72 byte (cf. Section 4.3). When transferring this data out of data owners' control, it becomes subject to (overlay) routing, processing, and storage operations in the back-end infrastructure. Performing these operations outside the control sphere of users raises severe privacy concerns [29, 35, 38]. Ultimately, this results in a complete loss of control over own data [16, 18, 19, 22].

To overcome these concerns, one promising approach in related work is to attach *per-data item privacy policies* (also referred to as *sticky policies*) to data before it leaves the control sphere of the user [15, 18, 28, 30, 34] as depicted in Figure 1. Privacy policies are thus imposed by the data owner and are binding for all entities involved in handling the data in the back-end infrastructure outside the control of the data owner. More specifically, data is only allowed to be routed to, processed on, and stored at nodes in the back-end fulfilling the privacy policy imposed by the data owner. Coupling of data and policy ensures continuous availability of

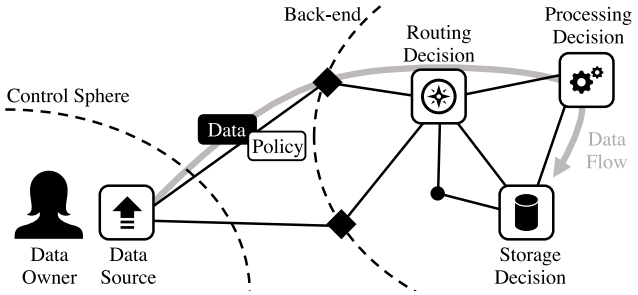

Figure 1: When data leaves the control sphere of the data owner, per-data item policies empower her to influence routing, processing, and storage decisions.

the policy. Alternative approaches such as per-stream policies [27, 37] lack support for the emerging federated clouds which distribute data among several cloud providers. Furthermore, existing data integrity protection mechanisms can easily be extended to the privacy policy to prevent inadvertent changes to privacy policies during transmission or data handling. In the context of this work, our aim is a functional improvement of the status quo by reducing policy sizes to feasible magnitudes for an interconnected world. We deliberately do not focus on the orthogonal problem of enforcing policies, i.e., providing (formal) guarantees that nodes adhere to policies. As CPPL does not change the semantics of policy languages, existing solutions that propose cryptographic guarantees [20, 23], tracking data flows [28], or creating audit logs [32] to enforce policies do still apply.

2.2 Requirements

The machine-readable formalization of privacy policies is called privacy policy language. In the following, we derive key requirements for any privacy policy language for the above described scenario where (potentially small) data leaves the control sphere of the data owner, e.g., in the context of cloud computing and the Internet of Things.

Minimal Storage Footprint. As privacy policies are attached to data and travel with it through the network, they add transmission and storage overhead. It is thus paramount that privacy policy languages minimize storage footprint.

Efficient Policy Checking. Privacy policies are evaluated at numerous times, e.g., relocation or replication and processing of data. Hence, the overhead for checking if a policy matches with the properties of a node must be minimized.

Expressiveness. We identify a large spectrum of expectations for handling of data: (i) restriction of storage location to a certain country, (ii) deletion of a data item at a specified point in time, (iii) logging or notification when data is accessed by a third party, or (iv) replication rate of data to ensure availability [15, 29]. A policy language must provide the ability to express expectations for these various kinds of data handling. This requires support of *environmental context*, e.g., awareness of storage location or replication rate, *time-based triggers* to specify the point in time for an action such as deletion, and *event-based triggers* to initiate actions when an event such as data access occurs [1].

Extendibility. In an interconnected world, new services and application scenarios together with novel privacy requirements emerge continuously. Thus, a policy language needs to be extendable such that it can be easily adapted to the individual requirements of new deployment domains.

Incremental Deployment. A new privacy policy language should be conceptually compatible with existing languages to integrate legacy deployments and ease transition.
Matching. A privacy policy language must support the matching between the privacy expectations of a user and what service providers offer. To this end, service operators must also be able to specify what their nodes technically provide and match this with user expectations.

2.3 Analysis of Privacy Policy Languages

In this section, we analyze (privacy) policy languages from related work with respect to our scenario and requirements. We summarize the results of our analysis in Table 1.

XACML [11] is a completely XML-based language for specifying access control policies. XACML is extendible to new requirements and use cases, but has an excessive storage footprint which requires applying separate compression [14]. Additionally, XACML has no support for triggers. *PPL* [6] and *A-PPL* [2,8] extend XACML with support of triggers, environmental context, credential-based access, and a matching procedure. *Henze et al.* [15] propose to extend PPL with triggers for storage duration and location.

Likewise based on XML, *PERFORM* [9] targets the scenario of pervasive computing. Policies in PERFORM specify actions as request/response pairs and limit these with the help of constraints. Its awareness of environmental context affords a good basis for expressiveness. However, it does not support triggers thus, e.g., not supporting access notifications or specification of data deletion at a certain date.

Rei [24] also targets pervasive computing and supports specification of rights, prohibitions, obligations, and dispensations. Expressiveness of policies profits from awareness of environmental context but lacks support for triggers thus facing the same limitations as PERFORM in our scenario. Furthermore, the size of resulting policies is not considered.

Garcia-Morchon et al. [13] propose an access control policy language for medical sensor networks. The resource constraints in this environment demand for a concise representation of policies. To this end, they specify policies in Boolean formulas represented as binary trees and efficiently stored in byte-level encoding. However, they explicitly focus on medical contexts which limits the generalizability of their language. Furthermore, matching of user expectations with provider-offers is unnecessary for their scenario but paramount in more general scenarios.

Ali et al. [1] describe an obligation language and a framework to enable privacy-aware service oriented architectures. Their language supports the specification of obligations, can evaluate the environmental context, and supports time- as well as event-based triggers. However, it misses a mechanism to match offers of a node with expectations formulated by a user [1], lacks an efficiency analysis, and does not consider a storage efficient representation of the formal language.

OSL [20] is a policy language for distributed usage control. In contrast to other languages, OSL partially supports the enforcement of policies by translating them into the DRM languages ODRL and XrML and then employing existing enforcement mechanisms. However, its performance remains unclear and no attention is paid to the storage footprint.

C^2L [32] is a highly specialized language for restricting the location and migration of virtual machines (VMs) in the context of cloud computing. A typed spatio-temporal logic enables enforcement of policies by rerunning the evaluation engine on the history of placement and migration of VMs. Hence, users are limited to a posteriori checking if a given history contradicts against a policy. Furthermore, the language is limited to the context of VM placement and thus does not provide sufficient expressiveness for the various applications in an interconnected world. Finally, matching of user expectations with provider-offers is not considered.

	storage footprint	efficiency	expressiveness	extendibility	deployment	matching
XACML [11] + extensions	∼		+	+	∼	+
PERFORM [9]	∼		∼		−	
Rei [24]	−		∼	+	−	
Garcia-Morchon [13]	+		∼		−	−
Ali [1]	∼		+		−	−
OSL [20]	∼		∼		∼	+
C^2L [32]	∼	∼	∼		−	−
S4P [4]	∼					+
FLAVOR [36]	∼		+		∼	−

Table 1: Comparison of existing privacy policy languages. A language fulfills (+), partially fulfills (∼), or does not fulfill (−) a requirement. Empty fields denote missing information.

S4P [4] focuses on matching privacy policies of users to those of service providers. To this end, S4P policies are specified in first-order function-less signatures and policies of users and service providers are compared using formal methods. This approach aims at realizing functionality and does not consider minimizing storage or processing overheads.

FLAVOR [36] focuses on legal rules which define consequences for infringements. To this end, FLAVOR does not only specify which policy a system should adhere to, but also which actions have to be taken if a posteriori verification detects a policy breach. FLAVOR's logic expressions enable specification of obligations with deadlines, triggers for external events, and context information. However, while focusing on a posteriori verification it does not consider matching of expectations and offerings of a service provider. Moreover, the storage overhead of a policy is not addressed.

To conclude, our analysis shows that no existing policy language supports all requirements for fine-grained privacy protection in an increasingly interconnected world. Most notably, existing languages either do not achieve a sufficiently small storage size to enable policies on item level or do not provide the necessary expressiveness and extendibility to cope with future, yet unknown privacy requirements. Furthermore, most existing works do not consider the importance of matching efficiency although it determines applicability for various upcoming scenarios, e.g., in the IoT.

3. INTRODUCING A COMPACT PRIVACY POLICY LANGUAGE

In an interconnected world, fine-granular user-centric privacy policies are a practical and much-needed solution for users to stay in control over their data. The main goal of this paper is to fill the identified gap between the requirements for privacy policies (Section 2.2) and the existing works (Sec-

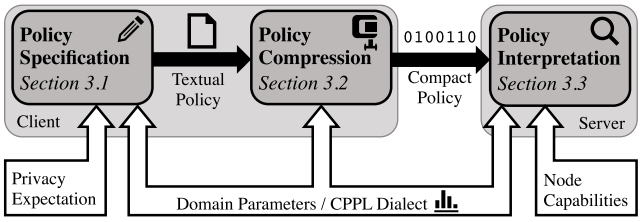

Figure 2: The core idea of CPPL is the compression of privacy policies through incorporating flexibly specifiable domain knowledge.

tion 2.3). Most notably, we require a high level of expressiveness and a minimal storage footprint at the same time.

To achieve this, we present CPPL, our **C**ompact **P**rivacy **P**olicy **L**anguage which relies on a two-step approach: First, a privacy policy is specified in a human-readable representation (as in related work). Here, we derive a policy representation that is as expressive as related work. In a second, novel step, we compress this policy by taking advantage of flexibly specifiable domain knowledge. Any further processing of the privacy policy, e.g., interpretation at nodes in the network, takes place directly on the compressed policy.

We depict an overview of our core design idea behind CPPL in Figure 2. Here, a user defines her privacy policy in a human-readable representation, possibly using a GUI or an editor. Our *policy compressor* uses this representation and a set of *domain parameters* to derive the compressed policy. The domain parameters define a *CPPL dialect* for a specific application scenario or deployment domain and define the variables and values that can be expressed in a privacy policy. Each dialect is specified by a central entity, e.g., a standardization organization. When interpreting a policy, CPPL uses the compressed policy, the domain parameters, and *node capabilities* of the node in question to evaluate whether the policy can be fulfilled by this node.

The design of CPPL has three main parts: (i) the specification of policies (in human-readable form), (ii) the compression of policies, and (iii) the interpretation of policies.

We provide a complete example of CPPL's specification, compression, and interpretation of policies in Appendix A.

3.1 Specification of Policies

For our specification of policies, we observed a common pattern in related work: Policies typically specify rules that list allowed (or forbidden) actions and individual rules can be combined using conjunction or disjunction. Hence, CPPL allows users to express their privacy policies as policy atoms (e.g., `location = "DE"`) which are connected via Boolean algebra. Our limitation to simple Boolean algebra is deliberate, since it enables even non-expert users to determine the meaning of a policy. However, CPPL is not inherently bound to this specification and can also work with other policies, e.g., XACML [11] and its derivates. Hence, with CPPL we do not propose a conceptually new policy language (in terms of what can be expressed) but rather show how to combine the concepts of existing policy languages with domain knowledge to achieve huge policy size savings.

We depict an example of CPPL's human-readable policy specification in Listing 1. In this example, data must not be stored at CompanyA, access to data must be logged, data has to be deleted after a certain point in time, backups have to be kept for one month, and the replication factor must be

```
  provider != "CompanyA"
& log_access = true
& deleteAfter(1735693210)
& backupHistory("1M")
& replication >= 2
& (   location = "DE"
    | (location = "EU" & encryption = true)
  )
```

Listing 1: Example of CPPL's human-readable policy that restricts co-location, location, and lifetime. It also enforces logging, backups, replication, and, depending on the location, encryption of the corresponding data item.

$$
\begin{array}{lll}
R & \rightarrow & \texttt{var}_{\texttt{bool}} \texttt{ ; !var}_{\texttt{bool}} \\
R & \rightarrow & \texttt{var}_{\texttt{num}} = \texttt{value}_{\texttt{num}} \texttt{ ; var}_{\texttt{num}} \neq \texttt{value}_{\texttt{num}} \texttt{ ;} \\
 & & \texttt{var}_{\texttt{num}} < \texttt{value}_{\texttt{num}} \texttt{ ; var}_{\texttt{num}} \leq \texttt{value}_{\texttt{num}} \texttt{ ;} \\
 & & \texttt{var}_{\texttt{num}} > \texttt{value}_{\texttt{num}} \texttt{ ; var}_{\texttt{num}} \geq \texttt{value}_{\texttt{num}} \\
R & \rightarrow & \texttt{var}_{\texttt{string}} = \texttt{value}_{\texttt{string}} \texttt{ ; var}_{\texttt{string}} \neq \texttt{value}_{\texttt{string}} \\
R & \rightarrow & \texttt{var}_{\texttt{enum}} = \texttt{value}_{\texttt{enum}} \texttt{ ; var}_{\texttt{enum}} \neq \texttt{value}_{\texttt{enum}} \\
R & \rightarrow & \texttt{func}(\texttt{par}_1, ..., \texttt{par}_N) \texttt{ ; !func}(\texttt{par}_1, ..., \texttt{par}_N) \\
F & \rightarrow & R \texttt{ ; !}F \texttt{ ; (}F\texttt{) ; }F \,\&\, F \texttt{ ; }F \,|\, F
\end{array}
$$

Listing 2: In CPPL's policy grammar, relations specify a comparison between variables and values, functions add support for triggers, and Boolean interconnections of these relations and functions (R) create a policy formula (F).

at least two. Furthermore, data has to be stored in Germany or, alternatively, in encrypted form in the European Union.

We depict the complete formal grammar of CPPL's human-readable policies in Listing 2 and in the following describe the important parts of this specification that we will later use to present an efficient policy compression in more detail.

Policy Atoms. Each CPPL policy is constructed out of different atoms, such as variables, relations, and functions. Properly differentiating between different types of atoms lays the foundation for efficient compression later on.

Variable Types: We differentiate between Booleans, numeric variables (integers and floats of different size), and strings. To ease compression of variables with a predefined set of values, we additionally support enumerations.

Relations: Boolean variables support negation (!), string variables can employ equality ($=, \neq$), and numeric variables additionally support ordering ($>, \geq, <, \leq$). A relation evaluates to *true* if and only if the comparison evaluates to *true*. This enables the comparison of expected (as specified in the policy) and actual environmental context.

Functions: Expectations with very flexible input such as event-based triggers, e.g., notification upon data access, and time-based triggers, e.g., performing backups within specific time frames or requiring data deletion at a specific point in time, cannot be expressed using relations in a scalable fashion. Hence, we support the specification of functions which consist of a function name and a list of parameters (e.g., `backupHistory("1M")`). Similar to relations, a function evaluates to *true* if and only if the node supports the expectation given by the function and its parameters.

Policy Formulas. We construct privacy policies out of the above relations and functions by interconnecting them with Boolean operations, i.e., and (&), logical or (|), and negation (!). To allow for concise formulas and increase readability, policy parts can be grouped with (nested) brackets.

Domain Parameters. To realize huge policy size savings, we, in contrast to related work, incorporate domain knowledge, i.e., what variables are available, which values they can take, and which functions can be utilized. This heavily depends on the individual use case. For example, the available variables might differ between a cloud and an IoT deployment. To this end, CPPL is parameterized to the individual use case through *domain parameters*, i.e., available variables, their type and value range, as well as available functions. Together domain parameters form a CPPL dialect which is provided by a central entity, e.g., a standardization body, for each domain.

For each variable, the specification states name and type, e.g., *Boolean*, *string*, or *int32*. Similarly, the available functions are also listed in the specification, together with the types of the function's parameters. For enumerations the specification lists all possible values. CPPL dialects solve three inherent challenges of policy languages: First, they provide users with a *list of possible requirements* they can specify in their privacy policies for a certain domain. Second, they enable *verification* of a policy, i.e., that it contains only valid variables and values. Third, they allow to *extend* the policy language to new demands in existing or new use cases. Notably, domain parameters are not defined by individual users and we expect them to stay rather static, occasionally being updated with a new version similar to the introduction of new versions of network protocols.

3.2 Compression of Policies

The centerpiece of our approach is the compression of privacy policies by taking advantage of specifiable domain parameters. To achieve a high compression ratio, we introduce the domain parameter specification to be able to incorporate domain knowledge into the compression step. The domain parameter specification lists the available variables, functions, and values in a well-defined order. This allows replacing variable and function names with a numerical identifier for their position in the domain parameter specification. A similar approach can be taken for enumerations.

A compressed CPPL policy consists of different parts. More precisely, we divide a compressed CPPL policy into four parts as illustrated in Figure 3: (i) the *policy header* stores an identifier for the domain parameter specification the policy relates to, (ii) the *formula stack* stores the logic operations connecting the relations of a formula, (iii) the *relation stack* encodes relation information, and (iv) the *variable stack* stores the numerical variable and function identifiers as well as actual values and parameters. With this separation we can leverage redundancies in privacy policies for compression. If a relation, variable identifier, or value is used repeatedly in a policy, we need to store it only once and can reference it repeatedly. In the following, we describe the encoding and compression of the parts of compressed CPPL policies in more detail.

Policy Header. To achieve a high compression rate, CPPL makes heavy use of information derived from the domain parameters defined in the used CPPL dialect. Hence, it is necessary to know a policy's CPPL dialect when interpreting the resulting compressed policy. As we cannot assume that this is always implicitly given by the context, we explicitly add the CPPL dialect in a 16 bit identifier field. As we strive for space efficiency, CPPL's policy header contains no further information. Notably, we completely waive

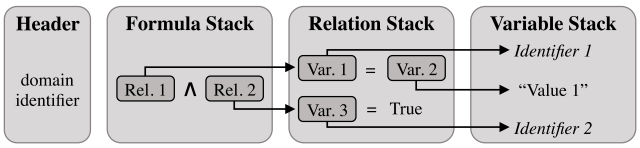

Figure 3: A compressed CPPL policy consists of a header and three stacks which reference each other to leverage redundancies for compression.

length fields (which are quite common for bit level encodings) as they introduce constant space overhead and put a limit on the overall policy size. Instead, we will show in the following how we encode lengths using special symbols and implicit knowledge derived directly from the encoded policy.

Formula Stack. We use a formula stack to encode interconnection of relations, i.e., logical operations, evaluation order as given by brackets, and references to relations. The overall goal of CPPL is to do this as space efficient as possible while still allowing for fast interpretation of the underlying policy. To save the space for an explicit encoding of the evaluation order, we rely on polish notation in the formula stack. While this automatically provides the correct evaluation order, we still require a space efficient encoding for combining references of relations using logical operations. To achieve this goal, we follow two paths: (i) reduce the number of logical operations that need to be encoded in the formula and (ii) reduce the space required for referencing relations.

We reduce the space for encoding logical operations by deferring the handling of negations to the relation stack through De Morgan's laws. Thus, we only need to differentiate between *and* and *or*, which can be encoded with only one bit. Alternatively, we can employ logic synthesis tools for hardware circuit design to minimize the size of the Boolean formula or optimize its representation for fast execution.

To reduce the space required for referencing individual relations, we order the relation stack according to the position of relations in the formula stack. Hence, we can omit references to relations in the formula stack and simply refer to the *next relation* on the relation stack. While this allows referencing relations very space efficiently, it prevents referencing one relation more than once (and thus saves space by leveraging redundancies). Hence, we introduce the concept of a *redundant relation* which allows referencing a relation that has already been used in the formula. The address of the referenced relation is specified in a fixed-size bit sequence[1].

Based on these optimizations, we only need to encode *and*, *or*, *next relation*, and *redundant relation* in the formula stack, which can be encoded with two bits. However, this does not allow us to signal the end of the formula stack (as discussed above, we do not use length fields for space reasons). To still be able to signal the formula stack's end, we introduce an additional bit to the *redundant relation* symbol to signal the end of the formula stack. Consequently, this adds an overhead of one bit to redundant relation identifiers (which is the least used of all four symbols).

Relation Stack. Similar to the formula stack, the relation stack encodes the interconnection of variable identifiers and variable values through relations. We use three bits

[1]We chose a fixed-size length to save overhead for a length field. This constitutes a trade-off between the number of relations that can be addressed and the space required for encoding references. As this trade-off is domain specific, we allow configuring this in the domain parameter specification.

to encode the relation types $=, \neq, <, \leq, >, \geq, = True$, and $= False$. Each relation type is followed by two respectively one (for $= True$ and $= False$) *next variable* and/or *redundant variable* symbols encoded in a single bit each. As for redundant relations, we add a fixed-size address to a variable on the variable stack after the redundant variable bit. In contrast to the formula stack, we do not explicitly signal the end of the relation stack as we can derive the number of relations on the relation stack from the formula stack.

Variable Stack. The variable stack encodes the variables (including functions) used in a CPPL policy. Each variable is represented by an encoding of its type followed by a type-dependent representation of the variable value. To encode the variable type, we differentiate between variable identifiers (where values are instantiated by the node interpreting a policy later on) and actual values (where values are already defined in the policy). We can derive all possible variable types from the domain parameter specification and encode them according to their order in the specification. Hence, the number of bits required for encoding variable types depends on the domain parameter specification. A reasonable set for variable types contains Booleans, integers (8 to 64 bits, signed and unsigned), doubles, strings, enumerations, and functions. Additionally, we reserve one encoding for variable identifiers. Hence, four bits suffice to distinguish between different variable types and variable identifiers.

The encoding of a variable type is followed by a type-dependent representation of the variable value as described in the following (encoding for new variable types can be easily deduced). First, variable identifiers are encoded as numbers as given by their order of appearance in the domain parameter specification. The number of bits required for this is determined by the number of variables in the specification. Boolean values are encoded as a single bit, integers and floats are encoded with their respective bit size, and strings are encoded as null-terminated ASCII values. When encoding numbers, we automatically use the smallest possible representation, e.g., a 32 bit integer will be automatically casted to an 8 bit integer if possible. For enumerations, we derive the encoding from the position in the sorted list of possible values for this enumeration. The variable type of an enumeration can be derived from the identifier of the variable the value in the enumeration is compared to. Finally, we encode functions by numbering their positions in the specification. Following the identifier for the function, we can directly encode the function's parameters, as their types are already defined in the specification.

Similar to relations in the relation stack, the number of variables in the variable stack can be derived from information in the relation stack. Hence, we do not need to encode the end of the variable stack and, thus, the end of a policy.

3.3 Interpretation of Policies

Once a CPPL policy has been compressed, it can be attached to data that is sent to other nodes. Each node that receives the data together with the policy interprets the policy, i.e., compares its own capabilities to the requirements in the policy. We first discuss these *node capabilities* in more detail before we present the actual *policy interpretation*.

Node Capabilities. The goal of privacy policies is to formulate requirements on the handling of data. This is predominantly achieved by comparing requirements to environmental context and supported triggers, i.e., the capabilities of a specific node [18, 30]. Only if the capabilities of a node match the requirements formulated by the user, this node is allowed to process the corresponding data. Essentially, node capabilities denote for each variable name in a domain parameter specification the values supported by this specific node. Furthermore, the node capabilities specify for each function defined in the domain parameter specification if it is supported by this node. If a node supports a function in general, the node uses a small script to check if it supports the parameters specified for this function as well.

Policy Interpretation. When interpreting a policy, i.e., deciding if a node supports the requirements in the policy, we replace the variable identifiers in the policy with the values listed in the node capabilities. To check if functions in the policy are supported, we extract the parameters and evaluate their support using the corresponding rules (see above). Finally, we evaluate the individual relations and then the complete Boolean formula. A node is eligible to process the data if and only if the Boolean formula evaluates to true.

During policy interpretation, we apply logical operations in the order given by the formula stack. This is possible, since the polish notation eliminates all brackets. When iterating over the formula stack to find the start of the relation stack, we sequentially push the operations onto a stack, obtaining reverse polish notation for the actual execution. Furthermore, we cache the result of each relation's interpretation to save processing time for redundant relations.

Additionally, a policy does not necessarily define an unambiguous handling of data, i.e., there may be more than one satisfying assignment. To cope with this challenge, we employ backtracking based on cached evaluation results to derive the actual variable assignment that must be employed.

4. EVALUATION

To thoroughly quantify and evaluate the space and processing savings of CPPL, we implemented CPPL in C++ using the Boost libraries. We utilize Flex++ and Bison to automatically generate the scanner respectively parser to process CPPL's textual policies. Furthermore, we realize the domain parameters and node capabilities specification using JSON and parse them with jsoncpp.

We first perform synthetic benchmarks to get a thorough view on the performance and scalability of CPPL and then realize policies for real-world scenarios which allows us to compare CPPL to related work. Based on this, we study the large-scale feasibility of CPPL in two use cases: (i) storing millions of IoT messages and (ii) matching thousands of policies when performing machine learning in the context of big data. Finally, we revisit our requirements for a privacy policy language in an interconnected world (cf. Section 2.2) and discuss how CPPL fulfills these requirements.

4.1 Influence Factors on CPPL's Performance

Performance and scalability of CPPL are influenced by policy size and volume of domain parameter specification. To intensively quantify both the influence on performance and scalability, we perform synthetic benchmarks for which we utilize a local test setup that consists of a desktop-grade machine (Intel i7 870, 4 GB RAM, Ubuntu 14.04). For each measurement point, we performed 100 runs and depict the mean value with 99% confidence intervals. We do not consider the overhead for initialization (in the order of 1.2 ms for compression and matching), e.g., for loading and pars-

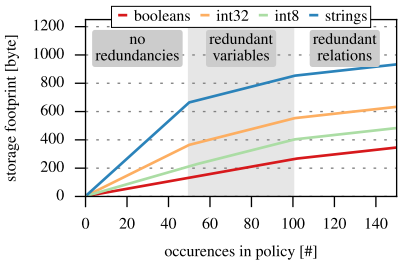

Figure 4: Policy size vs. storage footprint. Redundant variables, relations, and integer optimization improve compression.

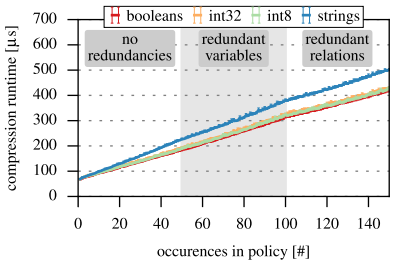

Figure 5: Policy size vs. compression time. Runtime scales linearly and variables with larger size increase runtime.

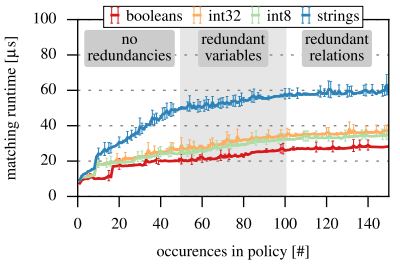

Figure 6: Policy size vs. matching time. Runtime increases for larger policies and strings benefit from redundancies.

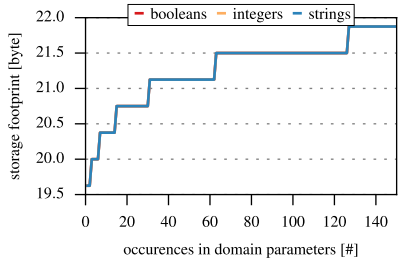

Figure 7: Domain parameter size vs. storage footprint. Increasing expressiveness increases storage footprint logarithmically.

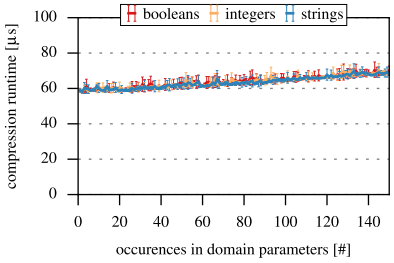

Figure 8: Domain parameter size vs. compression time. Tendency for a slight linear compression runtime increase.

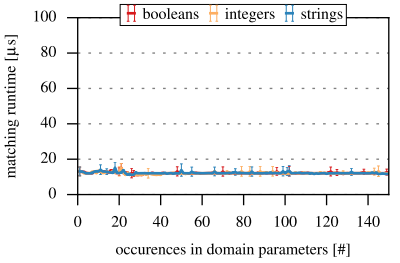

Figure 9: Domain parameter size vs. matching time. Matching runtime stays constant at a very low level.

ing the domain parameters specification, as this has to be performed only once when the system is started.

Increase in Policy Size. To evaluate the influence of policy size on storage footprint, compression, and matching runtime, we performed measurements with fixed domain parameters specifying 100 Boolean, integer, and string variables, each. We construct policies with up to 150 relations of the same variable type, allowing up to 50 (integer, strings) respectively 2 (Boolean) actual values. First, we explicitly evaluate a scenario without introducing redundancies for variables or relations (Relations 1 to 50). To study the impact of redundant variables, we then repeat already used variable values without repeating relations (Relations 51 to 100). Finally, to also study the effect of redundant relations, we duplicate the first 50 relations (Relations 101 to 150).

We use two integer sizes to evaluate CPPL's effect of automatically downsizing integers. While the domain parameters always specify integers with 32 bit, we used values whose representation requires 32 bit or only 8 bit in the policy.

We first depict the resulting *storage footprint* of a compressed CPPL policy in Figure 4. Without the possibility to leverage any redundancies, CPPL's policy size scales linearly, e.g., when considering a 32 bit integer from 9 byte for 1 relation to 364 byte for 50 relations. When introducing redundant variables, we observe a compression gain for strings (ratio 3.53) and 32 bit integers (ratio 1.93). In contrast, 8 bit integers and Booleans do not profit from redundant variables as the identifier for redundant variables would also consume 8 bit. Redundant relations allow for a further compression gain regardless of the variable type, e.g., by a ratio of 2.31 for 32 bit integers. Finally, the smaller storage overhead of 8 bit integers compared to 32 bit integers highlights the advantage of CPPL's automatic integer downsizing.

Next, we present the *compression runtime*, i.e., the time for transforming CPPL's textual policy into its compressed representation, depending on the policy size in Figure 5. Policies are compressed with a linear influence of the policy size. The compression runtime increases from 68 μs for 1 relation to 418–431 μs (502 μs for strings) for 150 relations. To put these numbers into perspective, CPPL is able to compress 1 993 to 14 754 policies per second. Strings show slightly more overhead due to slower encoding and comparison in the redundancy search. Redundant variables or relations do not noticeably influence compression runtime.

Finally, we show the *matching runtime*, i.e., the time for matching a compressed CPPL policy against node properties, in Figure 6. Matching (which is performed in the back-end and thus typically more often than compression), happens faster than compression with a linear increase in runtime for larger policy sizes. Without the possibility to remove redundancies, matching time for strings (Booleans) increases from 9 μs (7 μs) for 1 relation to 50 μs (21 μs) for 50 relations. Matching times for integers are slightly higher than for Booleans. Especially for strings, we observe a benefit of removing redundancies, reducing processing for strings (Booleans) for 150 relations to 58 μs (28 μs). Hence, CPPL is able to process 17 126 to 134 048 policies per second.

More Comprehensive Domain Parameters. We now evaluate the influence of more comprehensive domain parameters, i.e., a larger variety of variables that can be used in a policy, on policy size and processing time. To this end, we use a static CPPL policy consisting of 1 Boolean, 1 integer, and 1 string relation. For each of these variable types, we increase the number of domain parameters from 1 up to 150. Here, we do not differentiate between different integer types as the actual values only appear in the (fixed) policy.

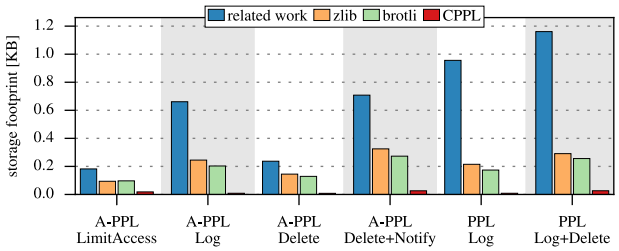

Figure 10: Real-world storage footprint comparison with related work. CPPL significantly reduces storage footprint compared to related work.

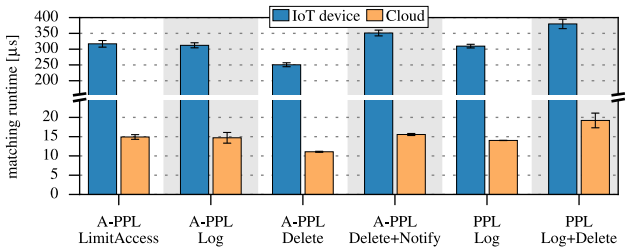

Figure 11: Real-world matching runtime for IoT and cloud class devices. Even on IoT devices, CPPL can perform thousands of matchings per second.

We depict the resulting *storage footprint* in Figure 7, where all three lines lie on top of each other, i.e., only the top most line is visible, as all variable types exhibit the exact same behavior. When the number of available parameters increases, CPPL requires more bits to encode variable identifiers, which is not affected by the variable type. We observe an increase from 19.63 byte for 1 variable definition to 21.88 byte for 150 variable definitions. More specifically, domain parameters that specify n variables require $\lceil \log_2(n) \rceil$ bits to encode the identifier in the variable stack.

When considering the influence of increasing domain parameters on *compression runtime*, we observe a tendency for a linear runtime increase in Figure 8. More precisely, the compression runtime increases from 59 μs for 1 variable to 70 μs for 150 variables. This results in approximately 14 288 to 17 004 policy compressions per second.

Figure 9 shows that the *matching runtime* is not influenced by increasing domain parameters (independent from the variable type). Matching runtimes are in the order of 12 to 13 μs. Consequently, CPPL is able to match 76 453 to 85 251 policies per second.

4.2 Comparison to Related Work and Real-World Performance

To verify the applicability of CPPL, we also evaluate CPPL on real-world policies taken from related work in the context of cloud computing and the IoT. To this end, we were able to locate 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 [8]) and two PPL policies (one specifying logging of three different actions and one extending the former with a deletion date [33]).

Comparison to Related Work. First, we analyze the required storage size for real-world policies. To this end, we compare the original XML representation (without superfluous whitespace) of the policy in A-PPL and PPL, respectively, with equivalent CPPL policies. As CPPL uses a compressed format, we also applied *zlib* and *brotli*, two compression libraries, to the policy representations from related work to also compare against generic compression methods.

We depict the resulting policy sizes in Figure 10. Overall, zlib and brotli achieve some compression, however, CPPL achieves by far the smallest size. For large policies, zlib and brotli achieve a compression ratio of 2.18 up to 5.49 while CPPL reduces the size by a ratio of 27.10 up to 112.47. For smaller policies, zlib and brotli perform worse, achieving a compression ratio of only 1.63 up to 1.94 while CPPL achieves a reduction by a ratio of 9.97 up to 29.63. In absolute numbers, CPPL 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. As we will see in Section 4.3, this results in an enormous reduction of the overall required storage space.

Real-World Performance. Finally, we evaluate CPPL's performance for the cloud and IoT domain. 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 in Figure 11 show that a cloud server can perform more than 52 056 policy matchings per second for our largest real-world policy. For smaller policies, this increases to more than 67 024 matchings per second. To put these numbers into perspective, even Dropbox had on average less than 20 000 insert/update requests per second in June 2015 [10]. For IoT devices, the matching rate still ranges from 2 632 up to 3 155 matchings per second. This is more than sufficient to process all messages in a deployed IoT platform (cf. Section 4.3), with the largest observed throughput of 149 messages per second. Thus, we enable policy awareness for the full data life-cycle from data collection by IoT devices to large scale processing in the cloud.

4.3 Large-scale Feasibility of CPPL

To demonstrate the feasibility of per-data item policies in general and CPPL in specific, we analyze the policy-induced storage overhead for data measurements in the IoT and investigate the impact of policy support for the runtime of machine learning approaches in the context of big data.

Storage Overhead in the IoT. The Internet of Things not only causes a massive growth in the amount of transferred data, e.g., up to 40 000 exabytes in 2020 compared to 130 exabytes in 2005 [12], but also significantly increases the diversity of data sources [3], and the granularity of reported data [21]. Hence, the question arises whether it is feasible to attach per-data item policies to IoT data as suggested by this paper and related work [15, 18, 28, 30].

To study the impact of per-data item policies on IoT data, we sampled frequency and size of real IoT data and analyze the storage overhead of attaching privacy policies to it. We collected real data of IoT devices from the API of dweet.io [5], a data sharing utility for the IoT. Our dataset, which we continuously collected over a period of 92 hours, consists of 18.41 million IoT messages originating from 7 207 distinct devices. The size of IoT messages ranges from 72 byte to 9.73 KB with a mean size of 394 byte. Although this data

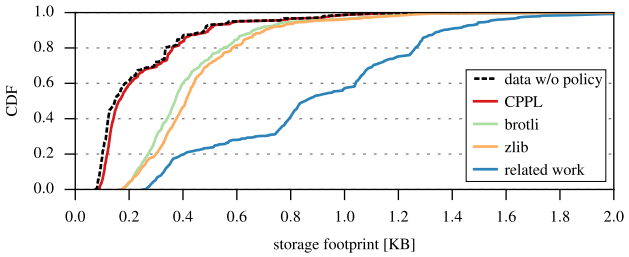

Figure 12: Impact of policies on storage footprint of real IoT data. CPPL policies significantly reduce storage footprint compared to related work.

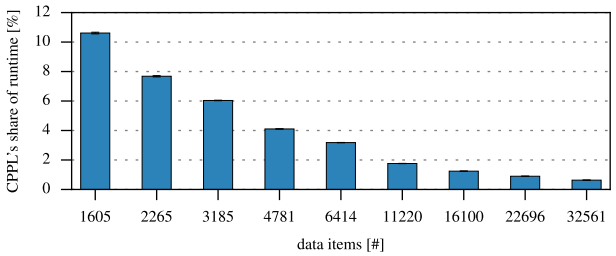

Figure 13: Impact of CPPL on machine learning (UCI Adult dataset [31]). For larger data sets, CPPL's share of the runtime becomes negligible.

is publicly available through dweet.io's API, we took appropriate measures to protect the privacy of people potentially monitored by IoT devices (data can, e.g., contain location information). To this end, we only stored the identifier of the IoT device and the timestamp of each data message. Furthermore, we sampled only one IoT message per device and solely stored the resulting message size (not the payload).

Figure 12 shows the cumulative distribution of IoT message sizes with (solid lines) and without (dashed line) attached per-data item policies. To this end, we uniformly randomly selected one of the policies from related work (cf. Section 4.2) for each IoT message and compare original uncompressed policies to policies compressed with zlib and brotli as well as our CPPL. These results show that CPPL adds only a negligible storage and transmission overhead compared to data without per-data item policy, while generic compression algorithms and especially uncompressed policies from related work induce significantly higher storage overheads. In total, 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. As this corresponds to less than 4 days of IoT data, it clearly highlights the necessity for space efficient privacy policy languages and the reasonable storage overhead that CPPL implies.

Policy Matching for Big Data. The massive growth in the amount of data is especially interesting for machine learning in big data which benefits from larger datasets for training models to increase their accuracy [26]. Per-data item policies can significantly increase willingness of individuals to contribute their data as these enable them to stay in control over their data. However, policies lead to additional processing for policy matching to determine if a policy allows usage of the data item for the desired application.

To investigate the performance of CPPL, we measure the overhead of policy matching when it is used to determine if data items can be used for a machine learning based study. We compare execution times of the training phase of the support vector machine LIBSVM [7] with the time required to process CPPL policies for this input data (we uniformly randomly assigned one of the policies from related work to each input and considered policy initialization overhead for the first occurrence of each domain parameters specification).

Figure 13 shows the share of the runtime that is required for policy processing for different numbers of input records of the UCI Adult dataset [31]. That is, the remaining share of the runtime is required for the actual training of the support vector machine (SVM). For a very small number of records, processing of policies takes 10.6% of the runtime that is required for the full process (policy processing and training of the SVM). More specifically, policy processing accounts for 18.9 ms while the training of the SVM requires 178.2 ms. However, with increasing number of data items used for SVM training, the fraction of time required for processing of policies considerably decreases. Considering, e.g., 32 561 data items, policy processing is responsible for only 0.6% (377.7 ms) of the total runtime whereas SVM training accounts for the other 99.4% (59.8 s). Hence, for larger datasets in the context of big data, the runtime overhead for CPPL policy processing is negligible. Thus, CPPL enables privacy policy-aware machine learning based approaches with almost no overhead on processing time.

4.4 Discussion of Requirements

We conclude our evaluation with a discussion on how we achieve the requirements that *any* privacy policy language in an interconnected world must fulfill (cf. Section 2.2).

Our benchmarks show that CPPL indeed achieves a *minimal storage footprint*, in which we significantly outperform related work. At the same time, our measurements illustrate that CPPL allows for *efficient policy checking* and is viable for real-world scenarios, especially at large scales. Furthermore, by reformulating existing privacy policies in CPPL we illustrate support for *incremental deployment* as it is compatible with existing policy languages. Through our concept of node capabilities in CPPL, we are able to realize *matching* of users' privacy expectations with the data handling offered by service providers. With our concept of domain parameters, we address the challenges of *expressiveness* and *extendibility* in CPPL. By combining Boolean expressions with run-time interpreted functions, we can cover all privacy requirements that are nowadays supported by related work simply by providing fitting domain parameters. Notably, domain parameters are what makes CPPL extendable: If additional privacy requirements in one of CPPL's application domains emerge, CPPL can easily be extended to support these by merely updating the corresponding domain parameter specification. Similarly, if completely new application domains emerge, CPPL can be effectively adapted to those by creating a new domain parameter specification (CPPL dialect). As this is done centrally by one entity, e.g., a standardization body, it does not place any burden on users.

5. CONCLUSION

In this paper, we presented CPPL, a Compact Privacy Policy Language for an interconnected world. CPPL allows users to specify their privacy requirements regarding rout-

ing, processing, and storage of data, e.g., in the context of cloud computing, the IoT, and big data. To this end, we follow a two-step approach: The owner of data first defines a policy in a human-readable representation (as with traditional policy languages). Then, CPPL compresses this policy, thereby optimizing policy size down to the bit level. To this end, CPPL takes into account the specific application scenario (e.g., cloud computing or IoT) and extensively utilizes domain knowledge to further reduce policy sizes. Still, our concept of domain parameters allows for easy adaption to new, even yet unforeseen use cases. CPPL further distinguishes itself from related work by its specific focus on the reduction of policy storage and processing overheads.

Our evaluation confirms that CPPL indeed drastically reduces policy sizes. Compared to related work, CPPL reduces policy sizes by up to two orders of magnitude, e.g., from 956 byte to only 8.5 byte. At the same time, CPPL can perform tens of thousands of policy matchings on a cloud server and still thousands of matchings on a tightly resource constrained IoT device. When considering the storage of massive amounts of IoT data, CPPL, in contrast to related work, adds only a marginal storage overhead. Likewise, CPPL can be used to express allowed purposes of data usage, e.g., in the context of large scale medical studies. To conclude, CPPL realizes a significant size reduction for privacy policies, which for the first time allows per-data item policies for fine-grained privacy protection in an interconnected world.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866. 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.

6. REFERENCES

[1] M. Ali et al. Obligation language and framework to enable privacy-aware SOA. In *DPM*. 2009.

[2] M. Azraoui et al. A-PPL: An accountability policy language. In *DPM*, 2014.

[3] P. Barnaghi et al. Semantics for the Internet of Things: Early progress and back to the future. *IJSWIS*, 2012.

[4] M. Y. Becker et al. A practical generic privacy language. In *ICISS*. 2010.

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

[6] L. Bussard et al. Downstream usage control. In *POLICY*, 2010.

[7] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM TIST*, 2011.

[8] R.-A. Cherrueau et al. Policy representation framework. Tech. report, A4Cloud Consortium, 2013.

[9] A. Dehghantanha et al. Towards a pervasive formal privacy language. In *AINA Workshops*, 2010.

[10] Dropbox Inc. 400 million strong, 2015.

[11] eXtensible access control markup language (XACML) version 3.0. OASIS Standard, 2013.

[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] O. Garcia-Morchon and K. Wehrle. Modular context-aware access control for medical sensor networks. In *ACM SACMAT*, 2010.

[14] D. Geer. Will binary XML speed network traffic? *Computer*, 2005.

[15] M. Henze et al. Towards data handling requirements-aware cloud computing. In *CloudCom*, 2013.

[16] M. Henze et al. A comprehensive approach to privacy in the cloud-based Internet of Things. *FGCS*, 2016.

[17] M. Henze et al. Moving privacy-sensitive services from public clouds to decentralized private clouds. In *IEEE IC2E Workshops*, 2016.

[18] M. Henze et al. The cloud needs cross-layer data handling annotations. In *IEEE S&P Workshops*, 2013.

[19] M. Henze et al. Towards transparent information on individual cloud service usage. In *CloudCom*, 2016.

[20] M. Hilty et al. A policy language for distributed usage control. In *ESORICS*. 2007.

[21] R. Hummen et al. A cloud design for user-controlled storage and processing of sensor data. In *CloudCom*, 2012.

[22] I. Ion et al. Home is safer than the cloud!: Privacy concerns for consumer cloud storage. In *SOUPS*, 2011.

[23] W. Itani et al. Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures. In *IEEE DASC*, 2009.

[24] L. Kagal et al. A policy language for a pervasive computing environment. In *IEEE POLICY*, 2003.

[25] P. Kumari et al. Distributed data usage control for web applications: A social network implementation. In *CODASPY*, 2011.

[26] S. Lohr. The age of big data. *New York Times*, 2012.

[27] R. V. Nehme et al. Fence: Continuous access control enforcement in dynamic data stream environments. In *CODASPY*, 2013.

[28] T. Pasquier et al. Data-centric access control for cloud computing. In *ACM SACMAT*, 2016.

[29] S. Pearson and A. Benameur. Privacy, security and trust issues arising from cloud computing. In *CloudCom*, 2010.

[30] S. Pearson and M. C. Mont. Sticky policies: An approach for managing privacy across multiple parties. *Computer*, 44(9), 2011.

[31] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods*, chapter 12. 1999.

[32] J. Poroor and B. Jayaraman. C2L: A formal policy language for secure cloud configurations. In *ANT*, 2012.

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

[34] S. Sundareswaran et al. Ensuring distributed accountability for data sharing in the cloud. *IEEE Trans Dependable Secure Comput*, 2012.

[35] H. Takabi et al. Security and privacy challenges in cloud computing environments. *IEEE Security Privacy*, 2010.

[36] R. Thion and D. Le Metayer. FLAVOR: A formal language for a posteriori verification of legal rules. In *IEEE POLICY*, 2011.

[37] C. Thoma et al. PolyStream: Cryptographically enforced access controls for outsourced data stream processing. In *SACMAT*, 2016.

[38] J. H. Ziegeldorf et al. Privacy in the Internet of Things: Threats and challenges. *Secur Commun Netw*, 2014.

APPENDIX

A. FULL EXAMPLE OF CPPL

In this paper, we presented the compression of a policy together with a reasoning of our design decisions. To fully embrace the inner workings of CPPL, we now present a detailed example for such a specification and compression of a policy as well as a description of its interpretation.

A.1 Specifying a Policy with CPPL

Listing 3 shows the textual representation of the policy which we will compress with the help of the domain parameters specification (CPPL dialect) given in Listing 4. The policy is an extended version of the policy discussed in Section 3.2 (see Listing 1) which, in the extended version, incorporates a redundant variable as well as a redundant relation to show the corresponding compression mechanisms.

```
  provider != "CompanyA"
& (   tenant != "CompanyA"
    | encryption = true
  )
& log_access = true
& deleteAfter(1735693210)
& backupHistory("1M")
& replication >= 2
& (   location = "DE"
    | (location = "EU" & encryption = true)
  )
```

Listing 3: Extended version of the previously used CPPL policy (Listing 1). The extended version features a redundant variable and a redundant relation to showcase their processing during compression.

A.2 Compressing a Policy with CPPL

The resulting compressed policy is depicted in Listing 5. The formula stack encodes the boolean operands *OR* (01) and *AND* (11). Furthermore, it refers to relations on the relation stack: either next relation (00) or to a relation at a specific position on this stack (011<*position*>). The *position* is specified as index of the relation on the relation stack starting with index 0 for the first relation. Thereby, the number of bits to encode the position of a variable is fixed and can be derived from the domain parameters (8 bits in our example). The end of the formula stack is signaled by the bit sequence 010.

Following the formula stack, the relation stack encodes the relations $=$ (000), $\neq$ (001), $<$ (010), $\leq$ (011), $>$ (100), $\geq$ (101), $= True$ (110), $= False$ (111). Thereby, it refers to one or two variables (depending on the relation type) on the variable stack: either to the next variable (0) or to a variable at a specific position on this stack (1<*position*>). Again, the position is given as the index of the corresponding variable on the variable stack starting with index 0 for the first variable. The length of the position field is specified by the domain parameters (here we use 8 bits which allows for referencing variables with index up to 255 — far more than required for the real-world policies in Section 4.2)

Finally, the variable stack encodes booleans (0000), variable identifiers that refer to variables specified in the domain parameters (0001), strings (0010), enumerated variables (0011), functions (0100), int64 (0101), int32 (0110), int16 (0111), int8 (1000), uint32 (1001), uint16 (1010), uint8 (1011), and double values (1100). Each of these type identifiers is followed by the actual value of the variable whose length is determined by the type, e.g., fixed to 8 bits for uint8 or terminated by a special symbol, e.g., for null-byte terminated strings.

This encoding enables us to reduce the 180 byte textual encoding to a 42 byte representation of the policy. This enables efficient transmission and storage of data annotations.

A.3 Interpreting a Policy with CPPL

When a node receives a data item, e.g., to store or process it, the node first must check if the desired action is possible within the boundaries specified by the policy. Compressed data typically requires decompression before its processing. However, with CPPL we are able to omit a separate decompression step and instead efficiently integrate decompression

```
{
  "version": 23,
  "relationPositionLen": 8,
  "variablePositionLen": 8,
  "variables": [
    { "name": "provider",
      "type": "string"
    },
    { "name": "tenant",
      "type": "string"
    },
    { "name": "log_access",
      "type": "boolean"
    },
    { "name": "deleteAfter",
      "type": "function",
      "parameters": [ "int32" ]
    },
    { "name": "backupHistory",
      "type": "function",
      "parameters": [ "string" ]
    },
    { "name": "location",
      "type": "string",
      "values": ["DE", "FR", "US", "GB",
                 "NL", "EU"]
    },
    { "name": "encryption",
      "type": "boolean"
    },
    { "name": "replication",
      "type": "int32"
    }
  ]
}
```

Listing 4: Domain parameters specification (CPPL dialect) in JSON format. The important content with respect to our example is highlighted in bold.

into the interpretation of the policy (cf. Section 3.3). In the following, we describe the interpretation of CPPL policies in more detail based on our example.

At the beginning of the compressed policy, the header enables the matching algorithm to determine the domain parameters specification (CPPL dialect) that applies to this policy. Following this, the formula stack encodes the boolean interconnection of relations in polish notation. During the matching process, the algorithm iterates over the formula

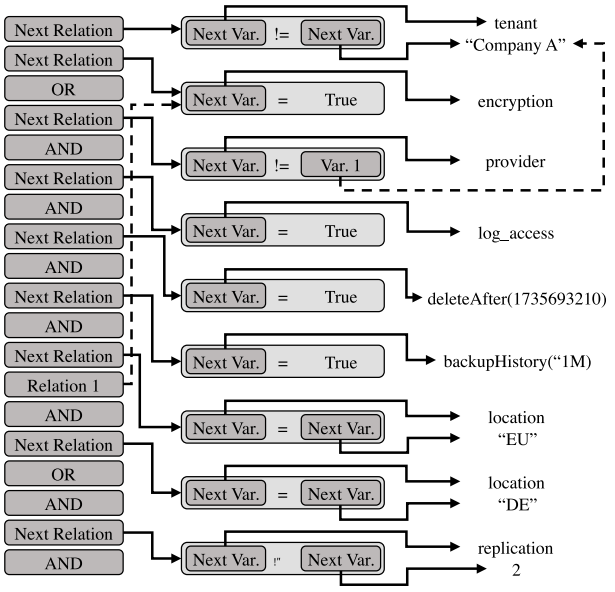

Figure 14: Decompression of a policy during matching. First, the algorithm iterates over the formula stack to find the beginning of the relations, thereby pushing elements of the formula stack onto an interpretation stack (left) which yields the policy in reverse polish notation. In a second step, the algorithm evaluates the policy based on the reverse polish notation, i.e., it resolves and evaluates relations and applies the boolean operations to the corresponding results.

stack until its end to find the beginning of the relation stack. Thereby, it sequentially pushes the content of the formula stack onto an interpretation stack. For our example, we depict this stack in Figure 14 (left). When reaching the end of the formula stack, the interpretation stack contains the policy in *reverse* polish notation. This order is used for the actual interpretation of the policy.

To this end, the algorithm sequentially takes the next element from the top of the interpretation stack. This element may be a reference to a relation or a boolean operand. The typical case for relations is a reference to the *next* relation on the relation stack. In this case, the algorithm locates the next relation on the relation stack, resolves corresponding variables, interprets the relation, and stores the truth value (values for variable identifiers are retrieved from the node parameters). In case of a reference to a *specific* relation (as identified by a relation position), we know that this relation has already been evaluated and the truth value can be reused (see reference to Relation 1 in Figure 14).

When retrieving a boolean operand from the interpretation stack, the algorithm can directly apply it as the reverse polish notation ensures that the corresponding relations already have been interpreted. Furthermore, reverse polish notation ensures that the last element of the interpretation stack is a boolean operand whose application yields the final result of the interpretation process.

As shown in Sections 4.2 and 4.3, this interpretation algorithm provides an efficient interpretation of policies rendering CPPL policies suitable for cloud and IoT scenarios.

```
Policy Header
  0000000000010111 version (23)

Formula Stack
  11          AND
  00          Next Relation
  11          AND
  10          OR
  00          Next Relation
  11          AND
  011 00000001
              Reference to Relation at
              index 1
  00          Next Relation
  11          AND
  00          Next Relation
  11          AND
  00          Next Relation
  11          AND
  00          Next Relation
  11          AND
  00          Next Relation
  10          OR
  00          Next Relation
  00          Next Relation
  010         End of formula stack

Relation Stack
  001 0 0     ≠, Next Var, Next Var
  110 0       =True, Next Var
  001 0 1 00000001
              ≠, Next Var, Reference to
              Variable at index 1
  110 0       =True, Next Var
  110 0       =True, Next Var
  110 0       =True, Next Var
  000 0 0     =, Next Var, Next Var
  000 0 0     =, Next Var, Next Var
  101 0 0     ≥, Next Var, Next Var

Variable Stack
  0001 001    ID 1 (tenant)
  0010        string
    01000011011011110110110101110000
    01100001011011100111100101000001
    00000000  "CompanyA"
  0001 110    ID 6 (encryption)
  0001 000    ID 0 (provider)
  0001 010    ID 2 (log_access)
  0100 011    Function, ID 3 (deleteAfter)
    01100111011101001001001110011010
              int32 (value: 1735693210)
  0100 100    Function, ID 4 (backupHistory)
    00110001010011010000000
              string "1M"
  0001 101    ID 5 (location)
  0011 101    enum value 5 ("EU")
  0001 101    ID 5 (location)
  0011 000    enum value 0 ("DE")
  0001 111    ID 7 (replication)
  1011 00000010
              uint8 (value: 2)
```

Listing 5: Compressed Policy. The policy is shown as a sequence of bits (bold) complemented by descriptive text for their respective meanings.