Towards DevOps for Privacy-by-Design in Data-Intensive Applications: A Research Roadmap

Michele Guerriero
Politecnico di Milano
Milan, Italy
michele.guerriero@polimi.it

Damian A. Tamburri
Politecnico di Milano
Milan, Italy
damian.tamburri@polimi.it

Youssef Ridene
BlueAge - Netfective Group
Bordeaux, France
y.ridene@bluage.com

Francesco Marconi
Politecnico di Milano
Milan, Italy
francesco.marconi@polimi.it

Marcello M. Bersani
Politecnico di Milano
Milan, Italy
marcellomaria.bersani@polimi.it

Matej Artac̆
XLAB
Ljubjana, Slovenia
matej.artac@xlab.si

ABSTRACT

With the onset of Big Data and Data-Intensive Applications (DIAs) exploiting such big data, the problem of offering privacy guarantees to data owners becomes crucial, even more so with the emergence of DevOps development strategies where speed is paramount. This paper outlines this complex scenario and the challenges therein. On one hand, we outline a tool prototype that addresses the key challenge we found in industry, more specifically, assisting the process of continuous DIA architecting for the purpose of offering privacy-by-design guarantees. On the other hand we define a research roadmap in pursuit of a more correct and complete solution for ensured privacy-by-design in the context of Big Data DevOps.

Keywords

DevOps, Big Data, Privacy-By-Design, Trace-Checking

1. INTRODUCTION

Nowadays the term Big Data is widely used due to the great potential behind this technology [6]. Generally speaking, Big Data refers to the huge and growing amount of data that technological environments today allow for continuous analysis. One of the critical aspects to be addressed in such Big Data is its privacy. Ensuring data privacy means determining what data can be shared with third parties and properly controlled by means of suitable privacy preserving policies. One of the key techniques which might succeed in providing privacy guarantees in Big Data, is Attribute-Based Access Control (ABAC), also called Granular Access Control [4]. ABAC goes beyond the traditional Role-Based Access Control (RBAC), as it allows access to critical resources by evaluating complex policies, that involve various attributes about who is going to be granted, with which kind of access and which resource is going to be accessed.

Stemming from the experience that we elicited as part of industrial action research within the BlueAge software factory at NetFective Technologies[1], in this paper we outline a model-driven prototype solution for enhancing the design of DIAs with privacy policies and their guarantee. Our approach consists of 3 continuous architecting steps: (1) allowing a developer to set ABAC access control policies on the architectural model (e.g. a UML component diagram) of a DIA; (2) propagating such policies using a model-driven pipeline; (3) monitoring their validity at runtime leveraging on trace-checking techniques. Although several approaches to model-driven role-based access control exist, e.g., SecureUML [7], we learned that a much more granular way of modeling access policies is required in the context of Big Data. Moreover, existing approaches need to be combined with state-of-the-art trace-checking technology [2] to enable quick and constant monitoring for the satisfiability of temporal-based access policies over very large system logs.

This paper offers a research prototype in pursuit of the above challenge and is the outcome of our initial investigations towards guaranteeing privacy-by-design of DIAs in a DevOps fashion.

In our initial experimentations we observed that, although with several limitations and strong assumptions, our working prototype shows promise in supporting the iterative and continuous architecting process of offering privacy guarantees, right from the earliest phases of the software lifecycle. Finally, discussing prototype assumptions, limitations and initial results, we conclude that the proposed solution can serve as a starting point for future research and development. However much more effort is still required. On these premises, we define a tentative research roadmap, by identifying some of the key challenges to be addressed in the future. The rest of the paper is structured as follows: in Section 2 we detail our research prototype while Section 3 illustrates its usage in a simple scenario. Section 4 outlines discussions of its structure, assumptions and limitations. Further on, Section 5 reports the research roadmap we identified and finally Section 6 draws conclusions.

ICPE '17 Companion, April 22–26, 2017, L'Aquila, Italy.

DOI: http://dx.doi.org/3053600.3053631

[1] www.bluage.com/company/netfective-technology-group

2. OUR RESEARCH SOLUTION: ARCHITECTURE OVERVIEW

Figure 1 outlines our solution architecture. The architecture comprises two main building blocks, a modeling environment and a runtime environment.

On one hand, at design-time a user (e.g. a QA or Security engineer) defines the architectural model of a DIA enriched with access control policies. The automated model-driven pipeline generates from the defined model a set of Metric Temporal Logic (MTL) formulae expressing the desired policies in terms of temporal constraints over system events. MTL [5] is a temporal logic with the ability to express metric, i.e. quantitative, timing requirements.

On the other hand, the generated MTL formulae are then deployed on Trace-Checking Service, which periodically checks over traces of events their validity, to ultimately monitor and report violations of the defined policies. To address these periodic checks in an automated fashion, in our research solution we leverage DICER [1], a DevOps tool introduced in our previous work, enabling the model-driven continuous deployment of DIAs on the Cloud. A default DICER-specific deployment model is inferred from the DIA architectural model using an ATL (ATLAS Transformation Language) model-to-model transformation, to automatically deploy our runtime environment, which consists of the Trace-Checking Service, along with all the Big Data platforms required to operate the DIA. Once the deployment is completed, the Trace-Checking Service is initialized with the DIA architectural model along with the generated set of MTL formulae.

Stemming from the architectural model, the Trace-Checking Service is able to identify and to locate all the information needed for building suitable traces, so that the installed MTL formulae can be checked over them using an appropriate trace-checker. The latter is an efficient large-scale trace-checker of MTL formulae introduced in [2], which allows us to quickly analyze very large traces and thus to quickly monitor and report violations of granular access policies.

2.1 Modeling DIAs with Granular Access Control Policies

In our scenario, a QA engineer who has to perform, among the others, privacy and security analysis of DIAs, can design, through the Modeling Environment, an architectural model of a DIA, conforming to the underlying metamodel, whose core part is shown in Figure 2. In this model a *ComputeNode* is represented as a black box containing a specific computation (e.g. a machine learning algorithm, a query, etc) which works on a set of input *Dataset*s to produce a set of output *Dataset*s. A *Dataset* is structured as a set of records with a number of *SchemaField*s. A *ComputeNode* can be implemented according to different processing types (e.g. batch, streaming) and using different Big Data middleware, e.g., Spark, Storm or Hadoop. Moreover the user can model the *DataSource*s providing the various Datasets. These can be *SourceNode*s, such as sensors or web pages, or *StorageSystem*s, such as Cassandra or MongoDB, which also provide a persistence service. A *ComputeNode* is owned by a *User*, which can plays different *Role*s (e.g. admin, analyst, etc) within the modeled system.

The user can then augment the obtained model with granular access control policies. A second extract of the metamodel focusing on the available concepts to define said policies is shown in Figure 3. All the concepts shown in Figure 2 are specializations of the generic *DIAElement* concept. A *Permission* associates its owner, which can be any *DIAElement*, with a set of *ActionType*s, or operations that are accessible on a protected element, which in turn is a *DIAElement*. The freedom coming from binding pairs of generic DIAElements makes the modeling language highly flexible, since, depending on the owner and the protected element, we can specify different types of access policies.

Permission includes two attributes for specifying its validity start and end times, giving the possibility to set time-based access control policies, a key feature of our framework. In our solution we refer to the specific case in which access is granted only within a certain time interval, even though the more general MTL language can express much more elaborated timing requirements.

Finally by setting *Property*s on DIAElements, in terms of ⟨key, value⟩ pairs, we can further characterize the owner of a Permission and its protected elements, making the policy more and more specific (i.e. granular). For instance a ComputeNode could have the Property ⟨*Location, Italy*⟩, which can be used to restrict the access based on the location of the node.

Once the modeling activities have been finished the user can activate the model-driven transformations pipeline. First, the DICER tool can generate a default deployment model conforming to the DICER modeling language. Essentially, DICER consumes the DIA architectural model to find all the modeling elements that require certain platforms to be available at runtime. For instance, if there are one or more ComputeNodes whose target technologies is Spark, an instance of the Spark execution engine has to be available at runtime in order to execute them. The same applies for each StorageSystem containing one or more Datasets. The output is a complete deployment model, with default configurations, representing the runtime environment as illustrated in Figure 1. DICER can at this point automatically deploy such model.

Second, once the runtime environment has been deployed, the next step is to initialize the Trace-Checking Service, which takes as input the designed DIA Architectural Model and the set of generated MTL formulae. The set of MTL formulae is generated at design time so that the user is able to validate or even refine it before it is sent to the Trace-Checking Service. The generation of the MTL formulae is done using a combination of a model-to-model and a model-to-text transformations, developed in ATL and Xtext respectively. This second model-to-model transformation produces, from the DIA Architectural Model, an *MTLModel*, i.e. a set of *MTLFormulae* conforming to the metamodel that we derived from the MTL grammar (Figure 4). Given a set P of atomic propositions (i.e. *MTLAtom*s), a *MTLFormula* is built from P using boolean connectives like *And* ($\wedge$), *Or* ($\vee$), *Not* ($\neg$), *Implies* ($\Rightarrow$), *IfAndOnlyIf* ($\Longleftrightarrow$) and time-constrained temporal operators like $\mathbf{U}ntil_{\langle T1,T2\rangle}$, $\mathbf{E}ventually_{\langle T1,T2\rangle}$ and $\mathbf{G}lobally_{\langle T1,T2\rangle}$. Depending on kind of connectives it is possible to define *MTLUnaryFormulae* or *MTLBinaryFormulae*. MTL supports the definition of intervals $\langle T1, T2\rangle$ constraining the semantics of the temporal operator on the basis of the specified time bounds $T1$ and $T2$. Intervals can be finite ($T1$ and $T2$ are constants) or infinite ($T2$ is ∞); and either open or closed, both on the left edge $T1$ and right edge $T2$. Informally, let ϕ and ψ be two

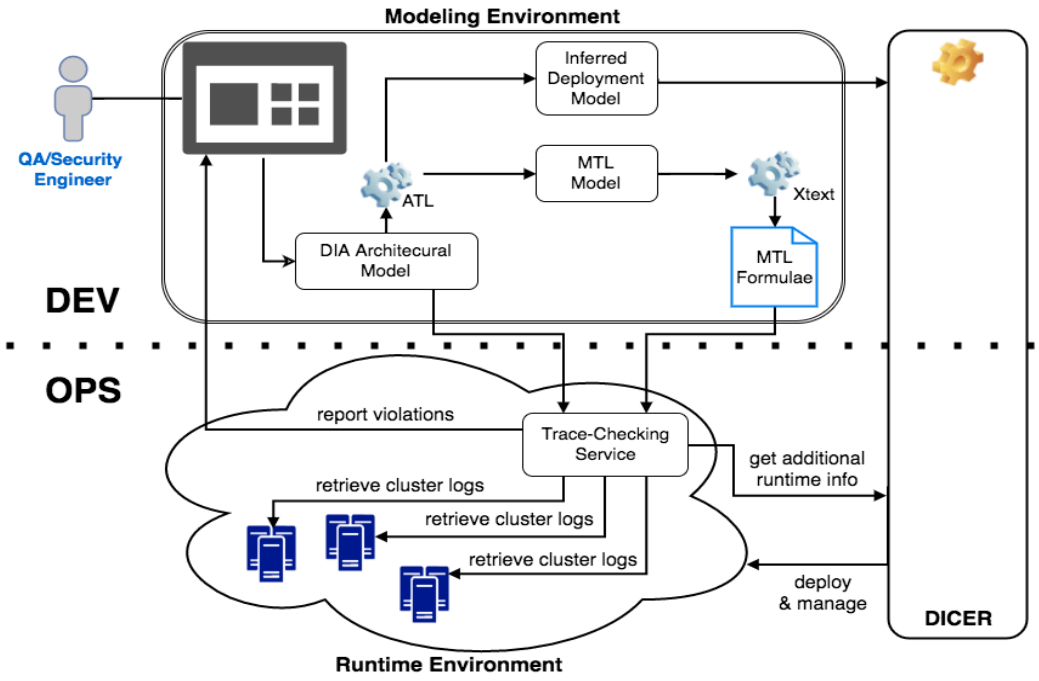

Figure 1: The overall architecture of the proposed system prototype.

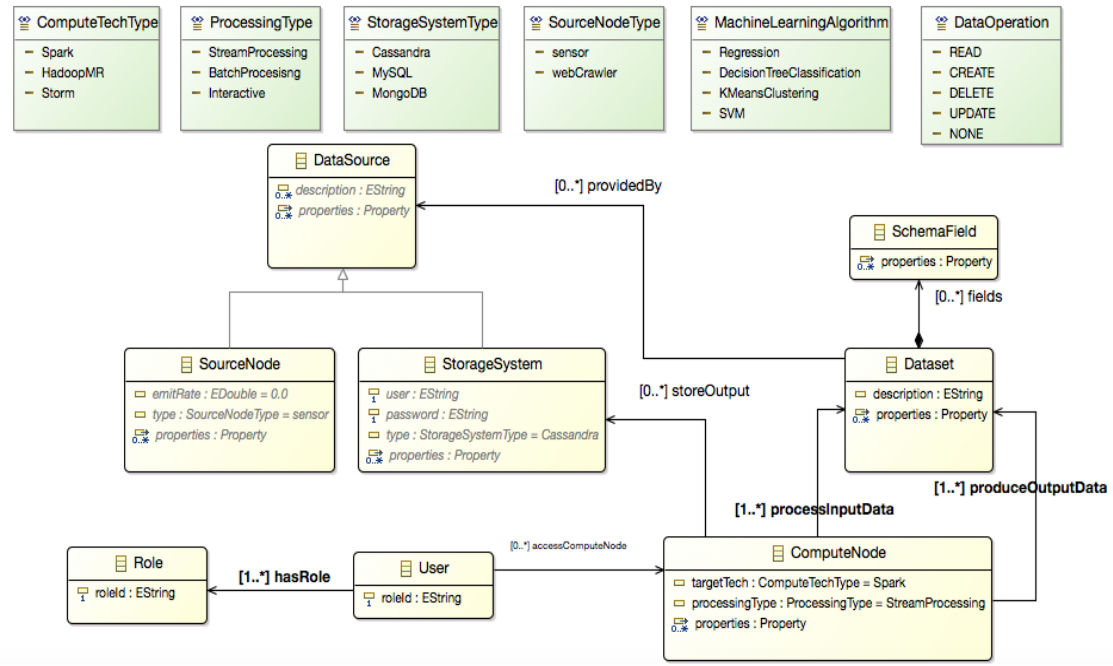

Figure 2: The core part of the metamodel grounding the modeling language used to create DIA architectural models.

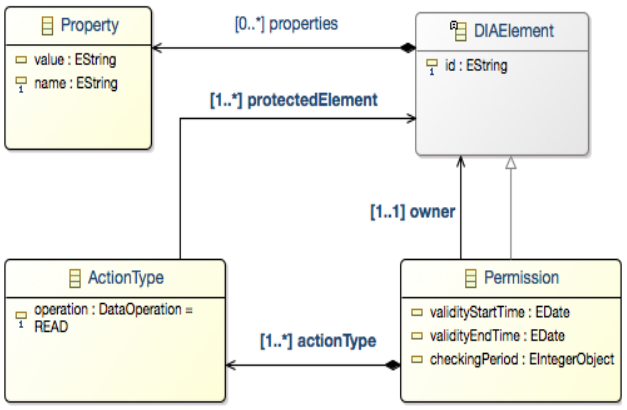

Figure 3: A focus on the metamodel capturing the main concepts to model access control policies.

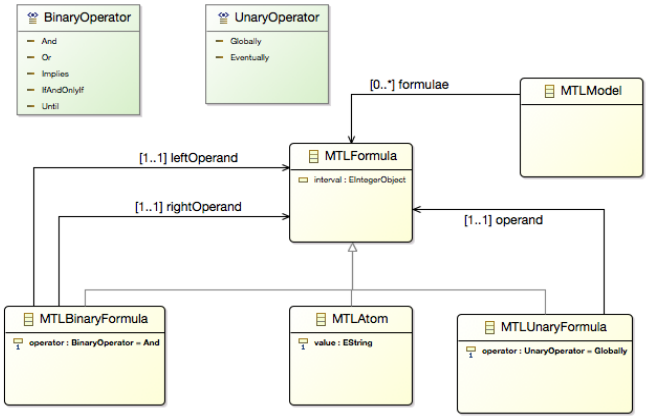

Figure 4: The metamodel used to describe MTL formulae.

MTL formulae. Formula $\phi \mathbf{U}ntil_{\langle T1,T2\rangle}\psi$ is true at time instant t when ψ holds at $t' \in \langle t+T1, t+T2\rangle$ and ϕ holds from t to t'. Formula $\mathbf{E}ventually_{\langle T1,T2\rangle}\phi$ is true at time instant t when there exists a time instant t' in $\langle t+T1, t+T2\rangle$ where ϕ holds. Formula $\mathbf{G}lobally_{\langle T1,T2\rangle}\phi$ is true at time instant t when, for all time instants t' in $\langle t+T1, t+T2\rangle$, ϕ holds. Specifying temporal bounds is a key feature to express time-constrained permissions in our approach. Each time value (i.e. $T1$, $T2$) represents a delay from the current instant in terms of the implicit time unit. For example, a property like "every time alarm goes off, the police is warned after (exactly) 30 seconds unless the security code is inserted first", considering the second as the implicit time unit, results as:

$$\mathbf{G}_{[0,\infty)}(Alarm \Rightarrow (\mathbf{E}_{[0,30)}SecurityCode \iff \neg\mathbf{E}_{[30,30]}WarnPolice))$$

There are different semantics and extensions of MTL. In this work we are considering MTL enriched with past operators, that is, temporal operators which predicate over the past. This will include operators like $\mathbf{P}ast_{\langle T1,T2\rangle}$ and $\mathbf{H}istorically_{\langle T1,T2\rangle}$ which are, respectively, the past version of $\mathbf{E}ventually_{\langle T1,T2,\rangle}$ and $\mathbf{G}lobally_{\langle T1,T2\rangle}$.

For each Permission instantiated in the DIA Architectural Model, the transformation generates a corresponding MTL formula. The current approach considers a single class of permissions, namely, each one of the form "the component X can execute operation O on the component Y only during the time interval from T1 to T2". Therefore, the transformation processes all the permissions in the same way, since only the validity time interval $\langle T1, T2\rangle$ defines the temporal property of a permission. The transformation generates the same MTL formula (i.e. the same syntactic structure) for each permission, except for the propositional atoms, which depend on the DIAElements involved in the permission. This decision is compliant with the requirements that we elicited from our industrial partners.

Finally the user can send the final DIA Architectural Model and the generated set of formulae to the running Trace-Checking Service.

2.2 What Happens at Runtime?

At runtime the Trace-Checking Service is responsible for monitoring and reporting back to the Modeling Environment access control policies' violations that might happen. Figure 5 represents the internal architecture of the Trace-Checking Service. By looking at the DIA Architectural Model, the Trace-Checking Service associates each permission (and its corresponding MTL formula) to a Driver process.

As a black box, the role of each Driver is simply to periodically update a *trace*, conforming to a specific format that can be read by the Trace-Checker, with necessary information for checking the satisfiability of the MTL formulae (and, in turn, the access policies) assigned to the Driver. Every time the trace is updated, the Driver also run the Trace-Checker to check if the MTL formulae are still satisfied. Finally, if there are violations, the Driver reports them to the Modeling Environment. The periodicity of the described process can be set by the user within the DIA Architectural Model. Each Driver conceptually represents the ability to check specific types of permissions. For instance a Permission granting a ComputeNode with the possibility to perform the READ action on a Dataset stored into a StorageSystem, can be checked as long as there is a Driver installed in the Trace-Checking Service dedicated to such kind of Permission (e.g. an application accessing a given dataset), which is able to retrieve all the necessary information from the available information systems (e.g. system logs, database tables, websites, etc). A different Permission could grant a User, instead of a specific application, with privileges for reading a specific SchemaField of a Dataset, instead of the Dataset as a whole. In this second case the Driver responsible for managing such policy at runtime will behave differently with respect to the first one, both in gathering all the necessary pieces of information and in building the trace. Moreover, the DIAElements involved in a given permission could be implemented using equivalent technologies (e.g. a streaming ComputeNode could be either a Spark or a Storm application), thus the responsible Driver should be realized in such a way as to be technology-agnostic and extensible.

If at some point the user wants to change a permission, she can re-traverse the model transformations pipeline and re-initialize the Trace-Checking Service. In this way our solution enables continuous architecting of DIAs in a flexible way.

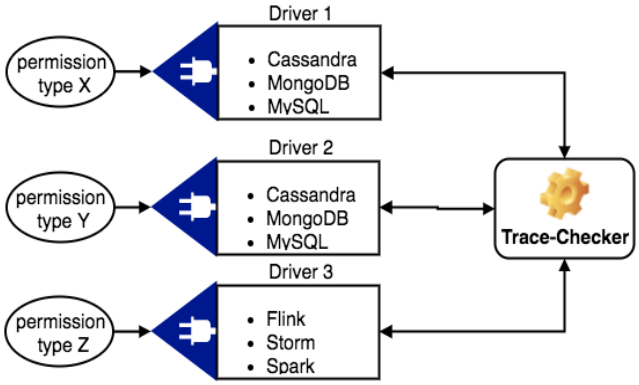

Figure 5: Representation of the internal structure of the Trace-Checking-Service

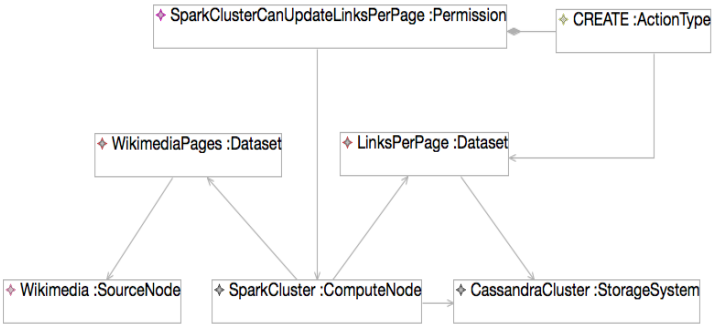

Figure 6: Example architectural model of a DIA processing web pages from Wikimedia, with an access policy in place.

3. EXAMPLE SCENARIO

In this section we are going to showcase our approach with a simple example, covering both the modeling aspects and the runtime behavior. As previously stated, the proposed modeling language is flexible enough to allow the definition of different kind of access policies, between different pairs of modeling elements. Each type of permission need to be managed appropriately by a specific Driver installed in the Trace-Checking Service, which also has to support the specific technologies involved in a given permission. In particular in our scenario we wanted to check permission of the type:"applications running on the middleware M can execute the CRUD operation O on the table T contained in the datastore D only during the time interval from T1 to T2". As we said, this is just one of the possible access policies that we could express. We focused on data access since this kind of access is particularly relevant in the context of guaranteeing privacy, that is the primary goal of our solution. Figure 6 shows the model obtained by instantiating the DIA architectural metamodel introduced in Section 2.1 for our scenario. In this example a single Spark ComputeNode (e.g. Spark applications running on a Spark cluster) accesses a Dataset of web pages from a SourceNode, that is the Wikimedia website, in order to perform various data analysis and produce aggregated data. For instance, a DIA could produce a Dataset called LinksPerPage, containing for each webpage its external references, that has to be stored into a StorageSystem, for instance a Cassandra cluster. The software architect decides, by modeling an appropriate Permission, that the Spark ComputeNode is allowed to perform the CREATE operation on the LinksPerPage Dataset only during the interval $\langle T1, T2 \rangle$. From such a model, the model-driven pipeline generates the following MTL formula:

$$Update(SparkCluster, LinksPerPage, CassandraCuster) \Rightarrow \mathbf{P}_{[T1,T2]} START \quad (1)$$

The Driver that is responsible for checking this specific type of data access permission also needs to include Cassandra among the supported datastores. In particular we found that Cassandra offers the possibility to enable a feature called *Probabilistic Query Tracing*, which essentially tracks the execution of each query. By enabling this feature we are able to periodically ask Cassandra for a time-ordered list of executed query, which also reports for each query the IP address of the VM from which the query was issued. We can then query the DICER service to resolve the cluster such VM belongs to, e.g., our Spark cluster rather than another distributed middleware available in the deployed Big Data infrastructure. By parsing the executed queries and resolving the query sender, our Driver is finally able to write traces like the one shown in Listing 1. By asking the Trace-Checker to verify the satisfiability of the MTL formula over the created trace, the Trace-Checking Service can successfully evaluate and report if the associated permission has been violated. In the provided example, assuming that (1) is instantiated with values $T1 = 15$ and $T2 = 20$, Listing 1 violates the permission as the update event with timestamp 22 happens outside the allowed time interval.

Listing 1: A piece of trace written for checking the example data access permission.

```
0 START
2 Elem(SparkCluster)
5 Elem(CassandraCluster)
7 Elem(WikimediaPages)
12 Elem(LinksPerPage)
15 Read(SparkCluster, WikimediaPages, CassandraCluster)
16 Update(SparkCluster, LinksPerPage, CassandraCluster)
22 Update(SparkCluster, LinksPerPage, CassandraCluster)
25 Create(SparkCluster, LinksPerPage,CassandraCluster)
30 Read(SparkCluster, WikimediaPages,CassandraCluster)
```

4. DISCUSSION

The proposed approach can be successfully used as a general DevOps framework for designing and prototyping privacy-aware DIAs by mean of Temporal-Based Data Access Control. In one of the possible scenarios, the framework would be used by a QA engineer who has to design and test privacy policies over DIAs. The framework could be also applied in production as a flexible way to monitor privacy-SLA violations. Moreover being constructed around formal and general trace-checking techniques, our solution is not limited to privacy and data access policies, but can be adapted to verify almost any runtime temporal-based property. Realizing a model-driven solution can be an effective way of enabling architectural level privacy analysis, in particular if it is supported with a DevOps cycle, allowing engineers

to quickly run and get feedback about the application in a production-like environment.

Some of the major limitations of our solution reside in its core engine, i.e., the Trace-Checking Service. The Trace-Checking Service is indeed a middleware requiring to access system-level runtime information, which turns out to be often unavailable, making difficult if not impossible to verify certain policies. This situation gets worse if we add the technological heterogeneity typical of DIAs. In our example we verified granular data access policies between middleware and datastores. In particular we used Cassandra as sample datastore. Replicating even the same scenario on a different technology could be not possible. For instance a different database could not provide the same information that Cassandra makes available by means of the tracing feature. Moreover, although we decided to apply MTL and trace-checking methods since we were interested in verifying temporal properties, this could be not the optimal approach for implementing the proposed solution, which might take also advantage from other powerful techniques, like for instance Complex Event Processing.

5. RESEARCH ROADMAP

In the previous sections we illustrated our approach to the problem of enriching the design of DIAs with the definition of privacy policies, and we highlighted a first prototypical solution, enabling the a-posteriori check of such policies by means of trace-checking techniques.

This section elaborates on the main challenges and possible next steps towards the improvement and extension of the presented approach. The goal is to obtain a more complete solution to guarantee ABAC policies for DIAs.

As highlighted in Section 4, one of the main limitations of our approach concerns the fact that policy violations can only be detected a posteriori by means of trace-checking. For this reason, a key improvement would be enabling the enforcement of ABAC policies. The problem is non trivial and needs further investigation.

Another possible improvement in that respect consists of adding some automatic mechanisms to deal with the detection of policy violations. For example, whenever a violation is reported by the Trace Checking Service, the system could "react" by adopting specific countermeasures, promoting the "continuous improvement" of the application.

One way to prevent violations is by enabling the secure deployment of DIAs. Our next step will be to support this approach at the DICER end. On one hand we will support the deployment of access policies at the data source level. On the other hand, we plan to support the deployment of secure datasets, meaning that private data are stored into protected datasets, that are accessible only by high privileged and trusted actors. The advanced version of DICER will produce deployment blueprints, that include actionable security and privacy policies.

A next major challenge is to support the whole DIA's development life-cycle, by extending the privacy analysis also to the design phase, applying, for instance, a model-based formal verification approach. Such extension would support the early detection of privacy-related design flaws and would foster the DevOps approach, by combining design-time and runtime analysis.

Furthermore, in order to at least partially overcome a major limitation of our prototype, i.e. the amount of information from multiple sources that it has to locate and retrieve at runtime, we plan to adopt specialised tools, such as the DICE monitoring tool [3], to collect and index some the necessary logs and data.

In conclusion, we argue that the problem of guaranteeing data privacy in the context of Big Data becomes critical and much more difficult to be addressed than in traditional data-intensive systems, mainly because of a) the huge amount of sensitive data that are captured and b) the new and complex technological landscape. This context requires new techniques to reason about data privacy, in order improve the design of privacy-aware solutions. On one hand, it is necessary to consider privacy issues from multiple perspective simultaneously, e.g. at the database and at the application level, and to adapt traditional privacy techniques to the current technological context. On the other hand, data privacy needs to be considered as a primary non-functional aspect of digital systems and privacy enhancing solutions must become much more pervasive than in the past.

6. CONCLUSION

This paper outlines a tool prototype to support the continuous architecting of privacy-aware DIAs. The prototype essentially allows designers to specify architectural models for their own data-intensive applications enriched with granular access control policies. Stemming from these models, our prototype is able to infer the necessary trace-checking "counterparts" for the specified privacy properties, expressed in MTL, a widely known metric temporal logic to convey properties over any given span of time. An expected and considerable benefit of the approach includes guarding sensitive data from exposure to malicious actors.

From our preliminary experimentation in industry we observed that the approach shows promise well beyond its current limitations. In the future we plan to follow the proposed research roadmap by enhancing the technical aspects and automations around the presented prototype, while instrumenting a more extensive evaluation campaign, both from an experimental and an empirical point of view.

Acknowledgments

European Commission grant no. 644869 (H2020 - Call 1), DICE.

7. REFERENCES

[1] M. Artač, T. Borovšak, E. Di Nitto, M. Guerriero, and D. A. Tamburri. Model-driven continuous deployment for quality devops. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, pages 40–41. ACM, 2016.

[2] M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krstić, and P. S. Pietro. Efficient large-scale trace checking using mapreduce. In *Proceedings of the 38th International Conference on Software Engineering*, pages 888–898. ACM, 2016.

[3] D. consortium. D4.1 monitoring and data warehousing tools - initial version. Technical report, IeAT, January 2016.

[4] G. D'Acquisto, J. Domingo-Ferrer, P. Kikiras, V. Torra, Y. de Montjoye, and A. Bourka. Privacy by design in big data: An overview of privacy enhancing technologies in the era of big data analytics. *CoRR*, abs/1512.06000, 2015.

[5] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[6] H. Liu. Big data drives cloud adoption in enterprise. *IEEE Internet Computing*, 17(4):68–71, 2013.

[7] T. Lodderstedt, D. Basin, and J. Doser. *SecureUML: A UML-Based Modeling Language for Model-Driven Security*, pages 426–441. Springer Berlin Heidelberg, 2002.