Security Policies Support for the reTHINK Web Service Architecture

Ana Caldeira
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: ana.caldeira@tecnico.ulisboa.pt

***Abstract*—The popularity of traditional operator-enabled services is decreasing as they are being replaced by Internet-based services, such as chat and social networks services. However, these services are often incompatible, which makes interoperability across them more difficult. To overcome this interoperability problem, the European Commission sponsored a research project named reTHINK, which aims to develop a web-centric P2P service architecture.**

For the development of the reTHINK project, the existence of mechanisms to specify and enforce security policies is fundamental. In particular, security policies are necessary to ensure authorized and secure access to resources. Considering a user scenario where it is possible to manage human context. Alice may find it interesting to allow her friend Bob to know her location at all times, whereas she does not want that with Trudy, someone who constantly disrupts her privacy.

The goal of this document is the description of a subsystem to support the specification and enforcement of security policies in heterogeneous application domains, where policies must be easy to manage and whose evaluation should be efficient. Concretely, this document presents PoliTHINK, a subsystem responsible for the specification, management and enforcement of security policies in the reTHINK framework. PoliTHINK offers low complexity in policies specification and good performance in their loading and evaluation.

I. Introduction

The rapid pace of Internet development allowed the emergence of new communication services to meet user needs. Standardization of such services facilitates the communication between clients of different service providers, but standardization is a complex process and tends to withhold the community of efficiently taking advantage of the Internet innovation potential. Consequently, the absence of standards causes a lack of interoperability, which is especially troublesome in applications with the same core functionality, for instance, messaging applications like Skype and WhatsApp, where both offer instant messaging services, but a WhatsApp user can only send messages within WhatsApp, not to a Skype user.

To overcome the announced interoperability problem, the reTHINK consortium [1] aims to develop a service architecture which enables authenticated and secure communication between applications from different service providers [2]. In reTHINK, the key to enable cross-service interoperability is based on *hyperlinked entities* (*hyperties*). Hyperties are reusable JavaScript code that implement the service logic of communication services, for instance, chat and audio. A hyperty works as a dynamically-loaded plugin that allows a user to communicate from any service provider he has subscribed, while guaranteeing interdomain communication by the means of a stub downloaded from the destination domain. Consider this example: Alice and Bob are two clients of chat applications sending a message to one another. Alice only has a Skype account and Bob only has a WhatsApp account. Nowadays, if Alice wants to send an instant message to Bob, Alice will have to create a WhatsApp account. Using the reTHINK Application, designed to support instant messaging communication between users from different service providers, the new hyperty concept eliminates that burden: the Chat Hyperty in Alice's device will coordinate the download of the software that contacts WhatsApp, hypothetically allowing her to send the message from her Skype account to Bob's WhatsApp account.

A particularly important component of the reTHINK design is the security policy subsystem, which is responsible for supporting the specification and enforcement of policies. Considering the aforementioned example, if for some reason Alice does not want Bob to reach her, the system must have a means for Alice to specify it and a mechanism to enforce it, i.e., block or redirect messages sent to her from Bob. This behavior, among others specified by the user or the service provider, requires a subsystem capable of interpreting and enforcing preferences and rules that manage access to resources.

This document presents PoliTHINK, a subsystem for the specification, enforcement and management of policies in the reTHINK framework. The design of PoliTHINK comprises several components: a formal language to be used in the specification of policies, an engine for their enforcement and a graphical user interface for their management. The design of the formal language provides an expressive language composed of a closed vocabulary to be combined in a way that covers all concepts the reTHINK project requires. The engine for policy enforcement has a modular architecture, which enables the reuse of the code in the reTHINK components that require policy enforcement. Its modularity is achieved by implementing all context-related dependencies separately of the core engine. Since

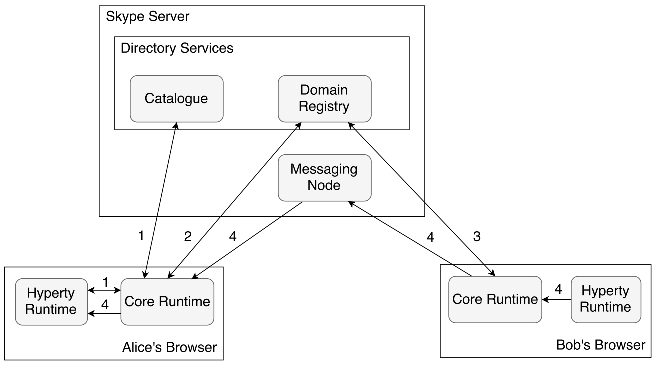


Figure 1. reTHINK framework (adapted from [4])

the user is not expected to understand a policy specification language, PoliTHINK provides a friendly interface for the browser where he can specify his reachability preferences. By the means of input fields, dropdowns and buttons, the user states what is the behavior he expects from the system, for instance, reject all incoming calls from 11 p.m. to 8 a.m.. The developed engine is currently integrated in two components of the reTHINK framework, the Core Runtime and the Vertx Messaging Node.

II. Background and Goals

A. The reTHINK Project

The main goal of the reTHINK consortium is to improve interoperability between services in the Internet of the future. In order to make possible the interoperability between such heterogeneous services, reTHINK relies on two new concepts: the *Hyperlinked Entity* and *Protocol-on-the-fly*. Both these concepts represent software components that are loaded in a dynamic and transparent way for the user device, and their role is to mediate access to existing services. Hyperlinked Entities (hyperties) are pieces of JavaScript code maintained by service providers to be instantiated in end user's devices. Since different service providers may use different communication protocols, in order to enable communication between them there are adaptations to be made. This is where the Protocol-on-the-fly is fundamental to resolve the interoperability issue: it dynamically selects and loads the implementation of the protocol stack of the destination service, which is then used to reach that service. It has a fundamental role on solving the interoperability among different services, as it promotes loosely coupled service architectures and at the same time minimizes standardization efforts [3].

To clarify these concepts, consider the following example, where Alice is using Skype as her service provider, and Bob is using WhatsApp. In Figure 1, Bob is trying to start a call with Alice. To achieve this, both needs to have an instance of a hyperty that provides a call service, available on the Catalogue of their service provider. The figure illustrates Alice's browser, Bob's browser, and Alice's service provider, Skype. In order to simplify the figure, it is only presented Alice's hyperty instantiation, which she does by contacting the Catalogue of her service provider (step 1), but the same process occurred for Bob by contacting WhatsApp. Then, a unique hyperty URL is generated by the Messaging Node, and that URL is registered along with the user identity in the Domain Registry (step 2), which enables Bob to be able to discover her by asking the Domain Registry which hyperty instances Alice has available (step 3). Having the hyperty instance address, Bob can publish a message through the Core Runtime to that address, which will be relayed by the Messaging Node (step 4) to Alice. The Core Runtime is comprised by a set of subcomponents responsible for the management of the hyperty instance execution in the user device, as it was introduced with this simple example. At the client side, both illustrated runtimes execute in the browser environment, in separate sandboxes.

The Core Runtime is a JavaScript middleware that runs in the browser and provides a secure runtime environment for hyperty and protocol stub code. The Core Runtime is also responsible for enabling local reTHINK applications to access the functionality delivered by co-located hyperties.

The Core Runtime is composed of a set of subcomponents, of which the most relevant for this overview are the Message Bus, Identity Module, the Policy Engine, the Runtime Registry and the Syncher Manager. The Runtime User Agent (Runtime UA) does the installation and management of these subcomponents. The Message Bus of the Core Runtime supports the communication attempt between the two hyperty instances illustrated in Figure 1, working as a pipe for messages exchanged between the service providers and the hyperty instance. The Identity Provider supplies the identity of the user that triggers the deployment of a hyperty instance in the Core Runtime, which is then stored in the Identity Module. For every message sent by a hyperty instance, the Identity Module is queried about which identity owns the hyperty instance, and this information is sent in the message to the destination so that the recipient can identify the sender. The Runtime Registry stores other relevant information about the hyperty instance and the Core Runtime subcomponents, and is available to be consulted by the remaining subcomponents. Communication between hyperty instances is done through the synchronization of objects distributed in the several runtimes. The Sync Manager subcomponent is responsible for creating those objects and for managing subscriptions that allow other hyperty instances to read it. Even though hyperties are controlled and coordinated by the user's runtime, we need to ensure that they behave according to the service provider specifications. Their correct behavior must be ensured by the Policy Engine, as a subcomponent that forwards or drops messages according to an authorization decision obtained from policies evaluation. Policies defined by the service provider are added to the Policy Engine by the Runtime UA when a hyperty is instantiated in the user's device. There is also the possibility to have user defined policies that concern user privacy and reachability preferences.

To enable a hyperty instance to contact another hyperty instance in a different device, we need some way of relaying the message to it. As illustrated in Figure 1, this is one of the responsibilities of the service provider Message Node. This component comprises a set of subcomponents, of which the most relevant for this overview are the Message Bus, the Address Allocation Manager, the Policy Engine and the Subscription Manager. Similarly to the Message Bus from the Core Runtime, the Message Bus of the Messaging Node works as a pipe for messages between hyperty instances and for messages used in their configuration. The Address Allocation Manager is responsible for the creation of a unique URL that identifies the hyperty instance in the reTHINK framework. For a hyperty instance to receive messages that are destined to it, the Subscription Manager adds and removes listeners for that address. The listeners intercept them and they are forwarded to the corresponding hyperty instance after validation in the Policy Engine, where service provider policies are enforced.

B. Goals and Requirements

The goal of this work is the design, implementation, and evaluation of a security policy subsystem for the reTHINK project. The security policies subsystem includes three fundamental components: a policy specification language, an engine to enforce the specified policies, and a graphical user interface to manage them. These components must be designed and implemented according to the following requirements:

- *Expressiveness of the policy specification language*, so that the language allows the representation of the concepts it is expected to (from characteristics of a message to user's events on his personal calendar),
- *Low complexity* on policy specification, so that the learning curve is as smooth as possible. This is accomplished by using intuitive keywords in the policies specification that directly correspond to the functionality they represent, and by providing a friendly user interface for policy management,
- *Efficiency* when evaluating policies, so that the introduced memory and processing time overhead is small,
- *Portability*, to allow its easy deployment on different environments, namely the browser and the message overlay system.

C. Contributions

The PoliTHINK subsystem makes four important contributions. First, it describes the design decisions of the reTHINK policy enforcement subsystem, which comprises several components: a formal language to be used in the specification of policies, an engine for their enforcement and a graphical user interface for their management. Second, it provides the implementation of the formal language and the engine in JavaScript, where policies are JavaScript objects that hold the representation of the behavior that is expected from the system. As the engine is independent from the context, it is deployable on both target platforms: the browser and the mobile environment. Third, it describes and exposes a crytical analysis of the evaluation of the implemented subsystem using quantitative tests. For instance, measurements done over a simple policy represented in our formal language show that its loading is done in approximately 1,8 microseconds and its evaluation in approximately 19,5 microseconds. Finally, PoliTHINK usability and correctness is confirmed by its integration on the reTHINK project in two different environments, the browser and the message overlay system. The integration in both components was possible due to the modular architecture of the subsystem, which allowed the code reuse.

During the development of both the design and implementation phases, the requirements previously defined were met, leading to an efficient solution completely integrated in two components of the reTHINK framework, the Core Runtime and the Vertx implementation of the Messaging Node. Also, this work resulted in a contribution to a conference paper [5] for the 19th International ICIN Conference - Innovation in Clouds, Internet and Networks.

III. Architecture

This section presents PoliTHINK, a subsystem for the specification and enforcement of policies in the reTHINK framework. In reTHINK, there are two entities involved in the specification of policies: the user, who uses hyperties to communicate with other users, and the service provider, who provides the hyperties and facilitates the communication signaling between the users. The user should have the opportunity to control his reachability preferences: who can reach him, when he can be reached, and what are the characteristics of the service being used to reach him. This is achieved by enforcing user policies in the Core Runtime of the user's device. On the other hand, the service provider needs to ensure the correct execution of hyperties in the user's device. To do so, the service provider needs, for instance, to guarantee that communication attempts from untrusted domains are denied, and to control the subscription configurations of hyperties by allowing or denying them to be subscribed by other hyperties. This is achieved by enforcing service provider policies in both the Core Runtime and the Messaging Node.

Consider Figure 2, which illustrates an overview of security policies in reTHINK. Since the service provider is expected to have enough programming knowledge to understand a policy specification language, adding policies to the system can be done by directly using the language

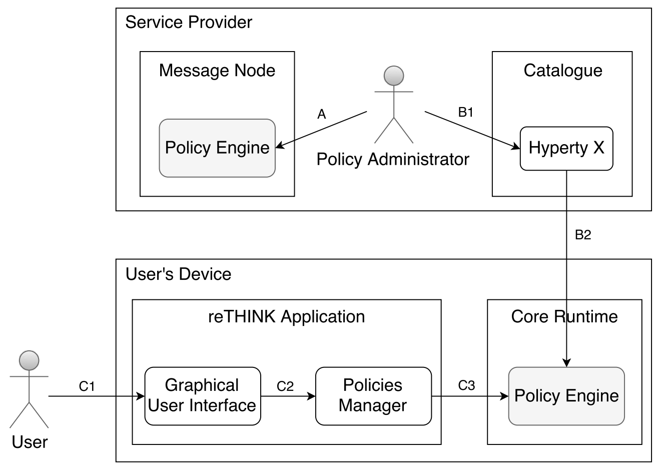


Figure 2. reTHINK Policy Specification Overview

```
policy: {
  key: <id>,
  rules: [{
    scope: <'global' || 'hyperty' || 'identity'>
    target: <'global' || hyperty-name || user-url>,
    condition: {
      attribute: <keyword>,
      operator: <operator>,
      parameter: <string || integer || array>
    },
    decision: <boolean>,
    priority: <integer>
  }],
  actions: [<method>],
  combining-algorithm: <algorithm-name>
}
```

Figure 3. The reTHINK Policy Syntax

for their specification. In the Messaging Node (situation A) this is done by feeding the Policy Engine with a policy file. These policies are independent of user IDs and hyperties, i.e., they are to be applied to all the intercepted messages. If a service provider wants to have policy enforcement for communications associated with a given hyperty in the user endpoint, he sends those policies in the hyperty descriptor (step 1 of situation B) to be enforced in the Core Runtime. When a hyperty is downloaded to a device (step 2), the policies retrieved from the hyperty descriptor are added to the Policy Engine. These policies are from then applied to messages regarding communications where the downloaded hyperty intervenes.

The user can specify his preferences by defining his own policies. Since it is not expected that the common user understands how to use a policy specification language to set his preferences in reTHINK, a graphical user interface is provided for this purpose. By the means of input fields, dropdowns and buttons, the user states what is the behavior he expects from the system, for instance, reject all incoming calls from 11 p.m. to 8 a.m. (step 1 of situation C). This information is then forwarded to the Policies Manager (step 2), which serves as translator from the provided input to the policy specification language by creating a policy specifying the restriction. The policy is then added to the Policy Engine in the Core Runtime (step 3) to be enforced from then on.

A. *Policy Specification Language*

In reTHINK, the data flow is carried through messages exchanged between hyperties and data objects attempting to create, subscribe or update hyperties on behalf of users. To ensure the correct execution of hyperties, these attempts must be validated by applying the service provider's and user's policies to obtain an authorization decision. Often, more than consulting who is attempting to do a given action, it is needed to consult other attributes of the system. It is useful to filter action attempts not only because it is a given subject attempting it, but because he is attempting it under circumstances that are not allowed, for instance, accessing an object on a time slot reserved for that object's maintenance. The reTHINK Policy Specification Language is a language tailored to the needs of the reTHINK framework. Policies following this language use a closed vocabulary that gives the ability to configure user or service provider configurations in an expressive and flexible way.

Figure 3 presents the fields of a policy and the syntax of each of them. For user policies, the *key* is an identifier chosen by the user when the policy is created. More than giving a hint to the user about what the created policy concerns, the key is a unique string that prevents the creation of multiple policies with the same identifier. For service provider policies, the *key* is given by the hyperty name, which identifies the deployed policy at hyperty instantiation time. A policy has a set of rules, where each rule is composed of several fields with enough information to obtain an authorization decision: the *scope* and *target* of the rule, the *condition* under which the rule applies, the *decision* to represent the authorization decision in case the condition applies, and the *priority*, a relevant field for the first-applicable combining algorithm used to manage the evaluation order of policies. Each policy also has the *actions* field, where the policy administrator may represent any additional action to be executed after the policy evaluation, independently of the authorization decision. Lastly, the combining-algorithm field allows the specification of the algorithm that solves possible conflicts in the individual rules' results.

The presented syntax is not enough to specify a policy that expresses an authorization decision based on multiple system attributes. PoliTHINK supports the use of the *and*, *or* and *not* logical operators to combine a set of attributes verifications. This feature allows having more expressive rules, as they enable the policy administrator to cover more combinations of real-life situations in the reTHINK framework.

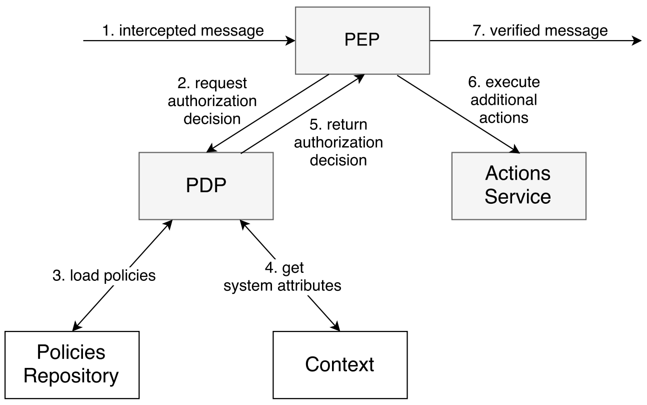
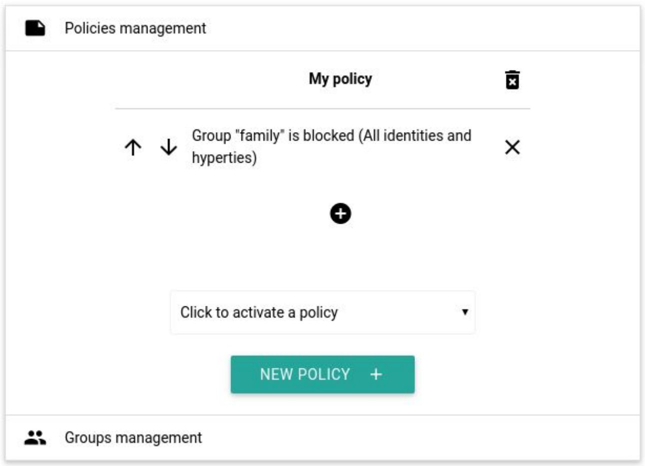


Figure 4. Policy Engine architecture



Policies management
My policy
Group "family" is blocked (All identities and hyperties)
Click to activate a policy
NEW POLICY +
Groups management

Figure 5. Policy administration page after creating the policy

B. Policy Engine

The Policy Engine is the reTHINK component responsible for validating the messages described in Section III-A. These policies can be specified by the user or by the service provider, and are enforced in two components of the reTHINK framework: the Core Runtime and the Messaging Node. In order to achieve Policy Engine's portability, one of the requirements of PoliTHINK, the context-specific functionalities are handled in a separate module, making the core of the Policy Engine a reusable module for any context in reTHINK. Since the evaluation of policies on both contexts only depends on the message content and on native JavaScript methods, both contexts can obtain all attributes that are necessary to enforce service provider policies. The main difference between them is on the additional responsabilities that the Core Runtime's Policy Engine has in the reTHINK framework: while the Messaging Node's Policy Engine exclusively validates messages against service provider's policies, the Core Runtime's Policy Engine has a much more important role in the management of hyperty execution in the endpoints. The Core Runtime's Policy Engine, more than an enforcer of user and service provider policies, also stamps the message with the identity of the sender to allow its identification at the destination, triggers the registration of hyperty subscribers in the Runtime Registry, and triggers the necessary procedures to ensure trustful communication between hyperties.

In order to enforce an authorization decision on a message, there are a few steps to perform when the message is intercepted. Consider Figure 4, which illustrates the Policy Engine architecture and represents the interactions between its subcomponents through arrows. When a hyperty posts a message in the Core Runtim, it is intercepted by the Policy Enforcement Point (PEP) before reaching its target component or leaving the runtime (step 1). Then, an authorization decision is requested to the Policy Decision Point (PDP) (step 2). The PDP generates the authorization decision by loading the existing user and service provider policies from the Policies Repository (step 3), which are verified by consulting the system attributes requested to the Context handler (step 4). The PDP returns the authorization decision generated by the policies evaluation to the PEP (step 5), and then the additional actions are executed (step 6) independently of the authorization decision. Lastly, if the message's evaluation results in a positive authorization decision, the message is forwarded to its destination; otherwise, the message is discarded. This architecture corresponds to the general Policy Engine, which consists of the components that are common to the different contexts it may be integrated in.

C. Administration Page

As previously introduced, the user is not expected to understand a policy specification language. Taking this into account, PoliTHINK offers a friendly user interface for the browser where the user can specify his preferences. The user interface is divided in two main sections: the first section provides the means to create, list, and delete existing policies and rules; the second section allows to create, list, and delete groups of users. It is possible to specify rules that restrict communication based on all attributes supported by the Policy Engine: the time of the day, date, weekday, source email, source domain, communication type, data object scheme and subscription preferences. Consider the following example: Bob had a huge fight with his sisters, Alice and Carol, and does not want to hear from them any time soon. To block all their contact attempts, she uses the reTHINK administration page to create the *family* user group, which has Alice and Carol contacts added, and creates a rule to block that group. These steps result in a new entry in the administration page that specifies that the *family* group is blocked for all identities and hyperties (Figure 5). When the new preference is submitted, it is translated to the policy specification language by the Policies Manager, which creates a policy using our formal syntax and uses the Policy Engine API to add the new policy to be enforced from then on. Adding to this simple approach, the administration page offers the option to import a policy file, which enables more advanced users to take full advantage of the policy specification language potential.

IV. Implementation

As a project for the web, the reTHINK framework is written in a scripting language based on JavaScript, ECMAScript 2015 [1]. This scripting language offers a powerful syntax which is very useful for the development of complex applications like reTHINK, for instance, objects classes and the Promise object. To be fully compatible with the reTHINK framework, PoliTHINK is itself implemented using ECMAScript 2015. The Policy Specification Language uses several classes that store the information about policies, rules and conditions. The enforcement of policies in the Policy Engine is managed by three main classes, one to obtain the attributes to be verified in the policy evaluation, another to generate an authorization decision based on the attributes obtained from the first, and another to effectively enforce that decision to a given message. The Graphical User Interface of PoliTHINK provides an administration webpage for the reTHINK application for policies, and is composed of two classes, one to handle the user interactions with the page, and another to translate the information provided by the user in the interface to our Policy Specification Language.

A. *Policy Specification Language*

For the specification of policies in reTHINK, it is fundamental to follow a fixed syntax across all environments to enable the correct management of policies. Therefore, a policy in reTHINK is organized in a fixed structure, similar to what can be found in the state of the art: each policy is comprised of a set of rules and a combining algorithm to resolve conflicts that may arise in the rules' evaluation. The supported combining algorithms are Allow Overrides, Block Overrides and First Applicable. The first gives priority to a positive authorization decision, the second to a negative authorization decision, and the third will prioritize whichever decision the evaluation of the first rule of the set decides (positive or negative). Among other information, each rule contains a Condition, which can be extended to a Subscription Condition or an Advanced Condition. In general, simple conditions verify one attribute of the system, subscription conditions verify the subscription preferences, and advanced conditions are able to combine several simple conditions through the *and*, *or* and *not* logical operators.

There are several attributes of the system that can be examined for rules' evaluation. Those attributes are represented in the *attribute* property of the Condition instance by a keyword that identifies it. The list of the keywords that correspond to each supported attribute is presented in Table I. These keywords were chosen because they represent the attribute they concern in a concise way, while enabling the policy administrator to intuitively understand which attribute they refer to. The *operator* property holds the function used to compare the current value of the system attribute specified in the *attribute* property with the *parameter* chosen, which can be one of the following: 'equals', 'in', 'greaterThan', 'lessThan' and 'between'.

[1]ECMAScript 2015 (or ES6) is standardized by Ecma International

Attribute	Keyword
Message source email	*source*
Message source domain	*domain*
Date	*date*
Weekday	*weekday*
Time of the day	*time*
Communication type	*type*
Data object scheme	*scheme*
Hyperties subscription preferences	*subscription*

Table I
Attributes of a condition and corresponding keywords

B. *Policy Engine*

The Policy Engine is responsible for the enforcement of the policies specified using our policy specification language. The enforcement of authorization decisions is done in the PEP, which are generated by the PDP by consulting the attributes retrieved from the class that implements the Context Handler; policies can specify additional actions to be executed, which are executed by the Actions Manager. To reuse the Policy Engine code, classes that work as plug-ins are implemented separately according to the specific environment the Policy Engine may be inserted in. The difference between the various contexts is mainly on how to load and store policies. Therefore, we implemented the Rethink Context class to provide the features that all reTHINK contexts can provide (namely time and fields of the messages circulating in the framework), and the particularities of policies management are handled by the context-specific classes, thereby enabling the portability of the component.

For the Policy Engine in the Vertx Messaging Node, which is implemented in Java, the Policy Engine Verticle class was created to work as interceptor of the Vertx internal bus, and the Policies Connector class as stub between the Java language and the JavaScript language, the programming language in which the Policy Engine is implemented. To be able to enforce policies in that environment, the Policies Handler class was created to intercept the messages circulating in the Message Bus.

C. Administration Page

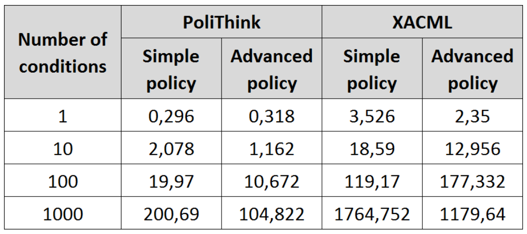
The User Interface is the graphical component of PoliTHINK that provides a simple way to specify policies for the average user. To build this interface, two classes were created: the Policies GUI, which handles the user interactions with the elements of the webpage, and the Policies Manager is responsible by translating those interactions to a policy in our policy specification language. To manage the creation of policies through the administration page, two classes were implemented: the Policies GUI and the Policies Manager. The first is responsible for handling the user interactions, i.e., dynamically builds the HTML elements necessary for the presentation and collection of the information, and collects the user input. The Policies Manager receives and organizes that information, which is then sent to the Policy Engine API and the policy instances. For a unified user experience, we used the Materialize [2] framework for the styling of the administration page, which was already being used for the styling of the reTHINK application.

V. Evaluation

This section presents the evaluation of the PoliTHINK system in terms of persistent memory usage, loading time and evaluation time of policies in the Core Runtime environment.

One of the goals of this document is the implementation of a solution that is efficient both in terms of memory usage and processing time. The time it takes to process a policy was separated in two stages: first, the loading time, second, the evaluation time. The tests to evaluate these variables will be performed for an increasing complexity of policies specified on our Policy Specification Language and also on XACML, the OASIS [3] standard for policy specification. To test the loading time of XACML policies, we used an XML parser provided by JavaScript. To test the evaluation time of XACML policies, we built a specific program to read the XML DOM tree in order to extract the information necessary for the verification of the conditions represented in the policy. To improve the reliability of the tests, each experience was performed 11100 times, distributed by 11 runs of 1010 repetitions each. From the 11 runs, the first one was ignored, and for each run, the first 10 repetitions were also ignored. This way it was possible to discard the warm up periods and equalize the obtained results, further improving their reliability.

The complexity of a policy varies, on the one hand, with the number of rules, and on the other hand, with the size of the condition in a rule. The evaluation of each metric will be done for two groups of policies. The first group is characterized by policies composed of 1, 10, 100 and 1000 rules, each holding a simple condition to be examined. The second group is characterized by policies composed of one single rule, with an advanced condition that combines the result of 1, 10, 100 and 1000 simple conditions through logical operators. When the reTHINK application is started, the Policy Engine is not populated with the policies that users or service providers may have specified in previous sessions. To populate it, these policies are loaded from the persistent memory and stored in a local hash table, which is used to retrieve policies for message evaluation in that session. The processing time of the two ways of loading policies will be evaluated, as well as the processing time of policy evaluation. These tests were performed in a machine with the Intel Core i3-2367M CPU @ 1.40GHz, 4 GB of RAM, running on 64-bit Ubuntu 14.04 LTS.



Number of conditions	PoliThink		XACML	
	Simple policy	Advanced policy	Simple policy	Advanced policy
1	0,296	0,318	3,526	2,35
10	2,078	1,162	18,59	12,956
100	19,97	10,672	119,17	177,332
1000	200,69	104,822	1764,752	1179,64

Table II
Occupied memory using the Persistence Manager (in kilobytes)

A. Memory Usage

The persistent way of storing policies in the Core Runtime is done through the Persistence Manager, which uses the Local Storage. It is a storage type that provides a means to store data with no expiration date within the user's browser. To understand the impact of loading policies from the persistent memory when the system is booted, we must quantify how many bytes their representation requires. Figure II presents the number of bytes occupied by the two groups of policies specified in both languages being studied. Taking into account the results obtained for PoliTHINK and for XACML, we observe that our solution is always more efficient than XACML in terms of memory usage. On average, a policy represented in PoliTHINK is 9,6 times more concise than XACML to specify the same behavior.

B. Policy Loading Time

To measure the loading time in the Core Runtime from both persistent and non-persistent memory, a timer is set using a library that generates the microseconds before the policy is loaded, and a second timer is set right after that request is fulfilled. Loading a policy with a given key from the Persistence Manager consists of requesting the Local Storage to retrieve the value that corresponds to the policy key, which is then parsed to JSON format. Subtracting the two timers gives the time it takes for the policy to be

[2]Materialize website: http://materializecss.com/

[3]OASIS (Organization for the Advancement of Structured Information Standards) is a global organization that works on the development and adoption of standards for security, Internet of Things, and other areas

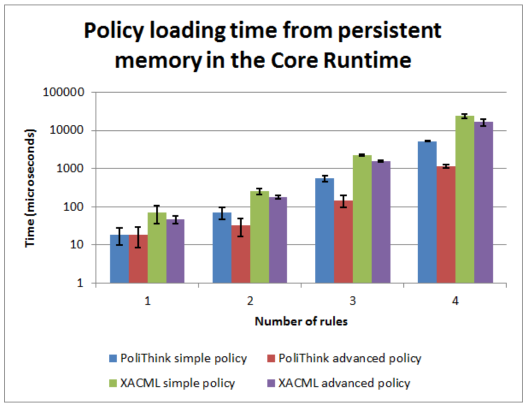


Figure 6. Policies loading time in the Core Runtime (in microseconds)

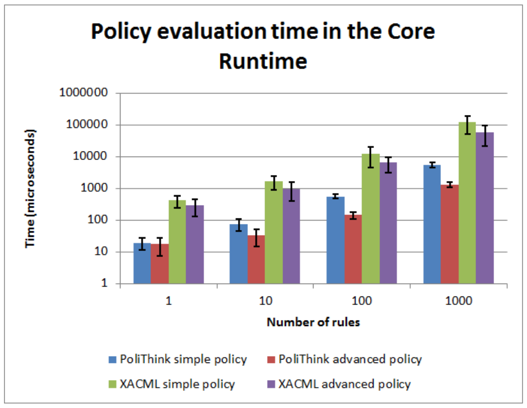


Figure 7. Policies evaluation time in the Core Runtime (in microseconds)

loaded from the Persistence Manager. The measurements obtained for the policies loading time in the Core Runtime are presented in Figure 6. Taking into account the results obtained for PoliTHINK and for XACML, we observe that our solution is always more efficient than XACML in terms of loading time from persistent memory. The policy loading time is closely linked with the corresponding memory usage, i.e., as the policies' complexity implies a higher number of bytes to represent them, the more time it is needed to load them.

To examine the performance of loading the policies from the hash table, a test similar to the one performed in the previous section was carried out: a timer was set before accessing the hash table, and a second timer was set after reading the policy that corresponds to a given key. Subtracting the two timers gives the time it takes for the policy to be loaded from the non-persistent memory. The measurements obtained for this test revealed that, on average, loading a policy from the hash table takes approximately 1,68 microseconds. This value corresponds to approximately 11 times less what was obtained for the best case of the persistent memory loading evaluation. To restrict the overhead introduced by the loading of policies to a minimum, each policy is loaded only once from the persistent memory when the session starts, and until the end of that session they are loaded from the hash table.

C. *Policy Evaluation Time*

The evaluation of a policy consists of evaluating each rule on the array of rules; the evaluation of a rule consists of verifying the condition applicability and returning the specified authorization decision if the condition applies. Consequently, the evaluation time of a policy is expected to be increasingly higher when the number of conditions to evaluate also increases. To measure the impact of policy evaluation, a timer was set before starting the evaluation of a policy, and a second timer is set right after the evaluation completes. Subtracting the timers gives the time it takes for a policy to be evaluated. The measurements obtained for the evaluation of the test policies are presented in Figure 7. Taking into account the results for PoliTHINK and XACML, we observe that our solution is always more efficient than XACML in terms of policy evaluation time. Such difference exists due to the means of extracting the information using each language. On the one hand, policies represented in the PoliTHINK Policy Specification Language are objects, and the policy properties are accessed directly. On the other hand, extracting the information from a policy represented in XACML requires traversing the node tree, which is computationally more expensive. Adding to this observation, and as expected, we observe that increasing the number of conditions to be evaluated results on a higher evaluation time.

VI. Related Work

In order to get some foundations about the specification and enforcement of security policies, a survey on the existing related work was carried out and presented in this section.

A. *Policy Specification Languages*

A survey on the most relevant policy specification languages was carried out, namely CSP [6], WS-Policy [7], XACML [8] and Ponder [9]. A table summarizing their expressiveness, specification complexity, verbosity and web compatibility - the policy specification language requirements for the subsystem to be developed - is presented in Table III.

CSP was shown to be a web-compatible and concise policy specification language, but its expressiveness is not enough for the reTHINK project needs. CSP only allows filtering sources of web page components (e.g. images and scripts), and the reTHINK framework requires a finer granularity to deal with the user's context and preferences.

WS-SecurityPolicy provides a more expressive syntax than CSP, but as a language designed to establish security

	Expressiveness	Specification Complexity	Verbosity	Web compatibility
CSP	Low	Low	Low	Yes
WS-SecurityPolicy	Medium	Medium	High	Yes
XACML	High	High	High	Yes
Ponder	High	Low	Low	No

Table III
POLICY SPECIFICATION LANGUAGES COMPARISON

constraints and requirements, it does not allow the management of the previously mentioned framework's resources.

Contrary to both CSP and WS-SecurityPolicy, XACML is a very expressive language. It supports the specification of reTHINK's policies, but at the cost of a high specification complexity and verbosity. Once the security policy subsystem efficiency is a fundamental requirement to fulfill, XACML's high verbosity (and consequent high number of bytes to be transferred, loaded and evaluated) makes it an unsuitable language to use in reTHINK. Furthermore, because it is a very general language, the specification of policies is a complex task that requires a specialized policy administrator, which also is not compliant with the reTHINK project, where the end user will have the opportunity to specify his preferences as policies to be enforced by the subsystem.

Similarly to XACML, Ponder is a very expressive language. Its advantage is related to the verbosity: it allows very concise specifications. Along with the low specification complexity, Ponder is shown to be the perfect fit for the reTHINK framework. Unfortunately, its implementation is no longer available.

B. Policy Enforcement Mechanisms

To enforce security policies over subjects attempting to perform operations on objects, accesses are generally controlled by reference monitors. The implementation of this abstract model has is characterized by properties that make the policy enforcement secure: it is non-bypassable, tamper-proof and evaluable. Some of the most relevant implementations of reference monitors for web content security were analysed, namely JSand [10], ConScript [11], WebJail [12] and XACML [8]. Both ConScript and WebJail are characterized by a deep approach, which demands client-side support in the JavaScript engine. The difference between the two is the granularity at which they operate: ConScript offers enforcement at web page level whereas WebJail offers the possibility of restricting the behavior of each component of the web mashup. JSand allows the website administrator to specify policies and have them associated with script providers at the client browser, whereas XACML allows for a complete decentralization of the creation of policies, the authorization decision and the enforcement of the decision. The security policies enforcement in the reTHINK project will take as reference the XACML model.

VII. CONCLUSION

This document presented PoliTHINK, a subsystem developed for the specification and enforcement of policies in the reTHINK framework that is divided into three subcomponents: a policy specification language, a policy enforcement mechanism (the Policy Engine), and a user interface for a more convenient way of specifying policies for the average user. The developed work is currently integrated in the reTHINK framework and is aligned with the requirements defined in this document.

REFERENCES

[1] reTHINK Consortium, "reTHINK - Trustful Hyper-linked Entities in Dynamic Networks," https://rethink-project.eu/, 2016, accessed: 2016-08-28.

[2] ——, "reTHINK Proposal," 2014.

[3] ——, "reTHINK D3.1 - Hyperty Runtime and Hyperty Messaging Node Specification," 2015.

[4] ——, "reTHINK D2.1 - Framework Architecture Definition," 2015.

[5] ——, "Global Identity and Reachability Framework for Interoperable P2P Communication Services," 2016.

[6] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10, 2010, pp. 921–930.

[7] W3C, "Web Services Policy 1.5 - Framework," 2007, accessed: 2015-12-07. [Online]. Available: http://www.w3.org/TR/ws-policy

[8] OASIS, "eXtensible Access Control Markup Language (XACML) version 3.0," 2013, accessed: 2015-12-03. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[9] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, ser. POLICY '01, 2001, pp. 18–38.

[10] P. Agten, S. Acker, Y. Brondsema, P. Phung, L. Desmet, and F. Piessens, "JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, 2012, pp. 1–10.

[11] L. Meyerovich and B. Livshits, "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, 2010, pp. 481–496.

[12] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen, "WebJail: Least-privilege Integration of Third-party Components in Web Mashups," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11, 2011, pp. 307–316.