Project Number 723634

D5.4 Full Prototype of the SPT Framework

Version 1.0
31 December 2018
Final

Public Distribution

The Open Group

Project Partners: **ATB, Electrolux, IKERLAN, OAS, ONA, The Open Group, University of York**

PROJECT PARTNER CONTACT INFORMATION

ATB Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de	**Electrolux Italia** Claudio Cenedese Corso Lino Zanussi 30 33080 Porcia Italy Tel: +39 0434 394907 E-mail: claudio.cenedese@electrolux.it
IKERLAN Trujillo Salvador P Jose Maria Arizmendiarrieta 20500 Mondragon Spain Tel: +34 943 712 400 E-mail: strujillo@ikerlan.es	**OAS** Karl Krone Caroline Herschel Strasse 1 28359 Bremen Germany Tel: +49 421 2206 0 E-mail: kkrone@oas.de
ONA Electroerosión Jose M. Ramos Eguzkitza, 1. Apdo 64 48200 Durango Spain Tel: +34 94 620 08 00 jramos@onaedm.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of York Leandro Soares Indrusiak Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325 570 E-mail: leandro.indrusiak@york.ac.uk	

DOCUMENT CONTROL

Version	Status	Date
0.1	Initial structure and outline	1 December 2018
0.5	Initial content	23 December 2018
1.0	Final version	31 December 2018

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

ABBREVIATIONS AND ACRONYMS

AD	Active Directory
AP	Abstract Platform
API	Application Programming Interface
App	Application
EP	Early Prototype
EPP	Event Processing Point
FoF	Factories of the Future
FP	Final Prototype
HW	Hardware
I&A	Identification and Authentication
IoT	Internet of Things
IIoT	Industrial Internet of Things
IIC	Industrial Internet Consortium
IIRA	Industrial Internet Reference Architecture
IISF	Industrial Internet Security Framework
Impl	Implemented
IT	Information Technology
LDAP	Lightweight Directory Access Protocol
NGAC	Next Generation Access Control
OE	Operating Environment (operating system)
OS	Operating System (operating environment)
PAP	Policy Access/Administration Point
PDP	Policy Decision Point
PEI	Policy Enforcement Interface
PEP	Policy Enforcement Point
PImpl	Partially Implemented
PIP	Policy Information Point
PM	Policy Machine
PQI	Policy Query Interface
RAI	Resource Access Interface
RAP	Resource Access Point
RI	Reference Implementation
SOPS	System of Production Systems
SPTM	Security Privacy and Trust Methodology/Mapping
SSF	SAFIRE Security Framework (or SAFIRE SPT Framework)
SW	Software
TBI	To Be Implemented
UC	Use Case
3PSM	Third Party Security Mechanisms

EXECUTIVE SUMMARY

This document describes the full prototype of the software implementing the functionality of the SAFIRE Security Privacy and Trust Framework as specified in deliverable D5.5 and implementing the methodology described in deliverable D5.1, thereby demonstrating the features and functionality of security services for the SAFIRE solution.

The document provides a description of the SAFIRE implementation of the NGAC standard, including the policy tool, the policy server, and the policy specification language. It also describes the integration with other modules and the functional architecture within which the components are to be deployed, and examples of how they can be used. Installation and Operation of the full prototype software are addressed, along with guidance for customisations for addressing diverse manufacturing applications where SAFIRE may be deployed.

1. INTRODUCTION

1.1 OVERVIEW

As introduced in D5.2 Early Specification of the SPT Framework, and finalized in D5.5 Final Specification of the SPT Framework, the SAFIRE security, privacy and trust effort intends to address two SAFIRE-relevant security challenges, and has two principal thrusts in SAFIRE. These comprise the SAFIRE Security Framework (or SAFIRE SPT Framework) (SSF):

- To address Security Challenge 1, that of being able to know the aggregate security policy of the various policies independently enforced, according to their configuration data, by distinct security mechanisms used within a complex system: An implementation of the Next Generation Access Control (NGAC) standard, that will provide the ability to specify aspects of policies corresponding to distinct segments or mechanisms within the system, and to compose those policy aspects to gain a unified and computable representation of the composed policy.

- To address Security Challenge 2, that of being able to assess the initial adequacy of the constellation of chosen security mechanisms and their respective configurations, and their continuing adequacy after a reconfiguration or optimisation has been applied: A comprehensive approach to security using the Industrial Internet Consortium (IIC) Security Framework (IISF) as guidance for the kinds of security functions that should be considered in a mission-critical industrial deployment of SAFIRE within a System of Production Systems (SOPS) Factories of the Future (FoF) setting.

1.2 APPROACH APPLIED

The first thrust, identified above, is realized by a software implementation of the NGAC standard that embodies one of the innovations of the SAFIRE project. The second thrust is realized by a methodology to assess the appropriateness and adequacy of security mechanisms chosen for a particular deployment, including those of the underlying platform as well as those provided by the NGAC implementation.

This document, D5.3 Early Prototype, describes the content of the first prototype of the NGAC implementation. The companion document, D5.1 Methodology, elaborates the methodology for the application of the IISF as well as methodology for the use of the software components of the NGAC policy specification and enforcement mechanisms.

The SAFIRE software portion of the SPT Framework includes:

- The policy tool (‘ngac’)
- The lightweight server (‘ngac-server’)
- A functional architecture for using the ‘ngac’ tool and ‘ngac-server’

1.3 DOCUMENT STRUCTURE

The document consists of:

- Section 1. Introduction—which describes the purpose of this document, and provided a brief overview of the contents of the document.
- Section 2. Description of the NGAC Full Prototype (FP)—our implementation of the NGAC standard, including the policy tool, the policy server, and the policy specification language.
- Section 3. Integration with other modules—describes the functional architecture in which the NGAC components are to be applied, and examples of how they should be used.
- Section 4. Installation and Operation—describes how to install and operate the NGAC FP components and how to invoke them from other components of the functional architecture.
- Section 5. NGAC Customisation—describes how NGAC can be customised for diverse applications.
- Section 6. Software Tools—identifies the software tools used for the implementation.
- Section 7. Conclusions and Plans.
- Section 8. References.
- APPENDIX A—Provides expanded background on NGAC.
- APPENDIX B—Provides a brief discussion of NGAC-based policy specification representations.

2. DESCRIPTION OF THE NGAC FULL PROTOTYPE (FP)

The functional architecture of NGAC as applied in the SAFIRE Project is depicted in Figure 1. We will refer to this figure frequently. Background on NGAC can be found in APPENDIX A – Next Generation Access Control.

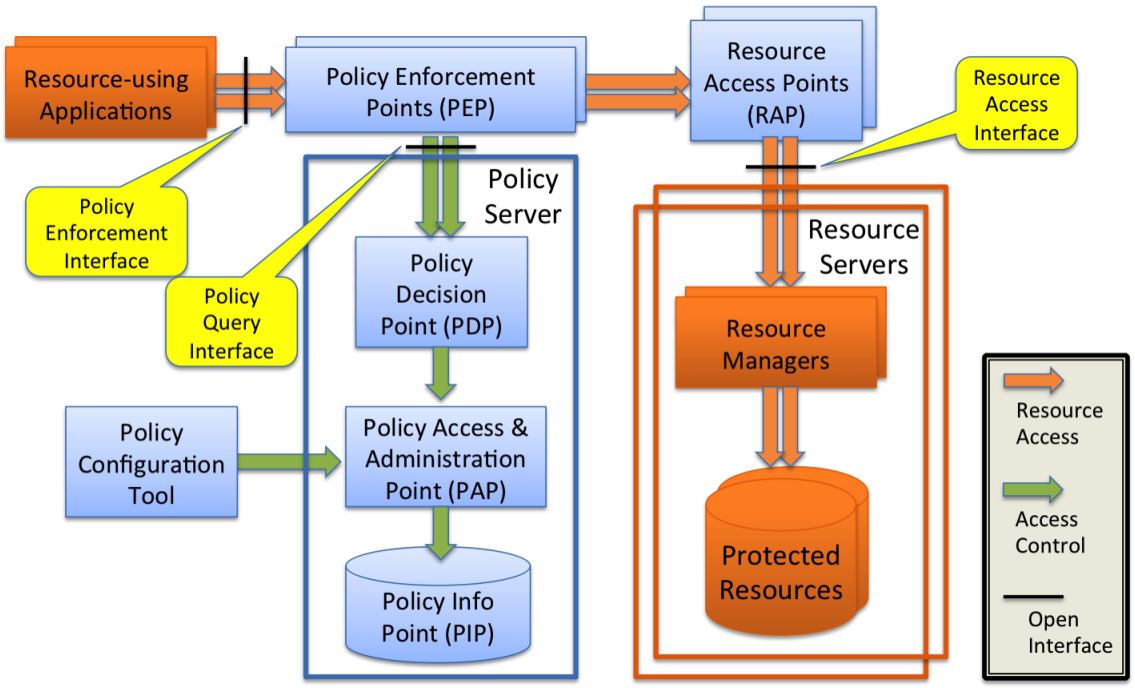


Figure 1: Our NGAC functional architecture with "unbundled" PEP & RAP

The components of the NGAC Full Prototype implementation for SAFIRE are:

- Enhanced 'ngac' policy configuration tool for doing standalone policy development and testing;
- Policy Server with RESTful APIs implementing the Policy Query Interface;
- Enhanced declarative policy specification language supporting modular policies and policy composition;
- Policy Access Point implementing the Policy Administration Interface and a Policy Information Point policy store, also provided within the Policy Server;
- Policy Enforcement Point template implementing the Policy Enforcement Interface; and,
- Resource Access Point template using the Resource Access Interface.

2.1 'NGAC' POLICY TOOL

The 'ngac' policy tool enables standalone policy development and testing and has the ability to start the policy server. The policy tool offers the command line prompt "ngac>", to which it accepts a number of administrator and advanced user commands.

2.1.1 Policy tool interactive commands

The 'ngac' Policy Tool is a command driven application. After starting 'ngac' it offers the prompt "ngac>". There are a set of basic commands available in the normal mode (admin) and an extended set of commands for use by a developer in development mode (advanced). Entering the command "help" will list the available commands in the current mode. Only the most commonly needed normal mode (admin) commands are introduced here.

- `access(` <policy name>`,` <permission triple> `).`
  Check whether a permission triple is a derived privilege of the policy.
- `admin.`
  Switch to admin (normal) user mode.
- `advanced.`
  Switch to advanced user mode.
- `aoa(` <user> `).`
  Show the user accessible object attributes of the current policy.
- `combine(` <policy name 1>`,` <policy name 2>`,` <combined policy name> `).`
  Combine two policies to form a new combined policy with the given name.
- `echo(` <string > `).`
  Print the argument string, useful in command procedures.
- `halt.`
  Exit the policy tool. (Will also terminate spawned server.)
- `help.`
  List the commands available in the current mode.
- `help(` <command name> `).`
  Give a synopsis of the named command.
- `import_policy(` <policy file> `).`
  Import a declarative policy file and make it the current policy.
- `newpol(` <policy name> `).`
  Set the named policy to be the new current policy.
- `nl.`
  Print a newline, useful in command procedures.

- `proc(` <procedure name> `[, step]).`
  Run the named command procedure, optionally pausing after each command.

- `proc(` <procedure name> `[, verbose]).`
  Run the named command procedure, optionally verbose.

- `regtest.`
  Run built-in regression tests.

- `script(` <file name> `[, step]).`
  Run the named command file, optionally pausing after each command.

- `script(` <file name> `[, verbose]).`
  Run the named command file, optionally verbose.

- `selftest.`
  Run built-in self tests.

- `server(` <port> `).`
  Start the policy server on the given port number.

- `version.`
  Display the current version number and version description.

- `versions.`
  Display past and current versions with descriptions.

There are, and may in the future be, additional advanced user commands to support development and diagnostics.

2.1.2 Command procedures and scripts

There are predefined 'ngac'command procedures ("procs") that run the examples and can be used for testing and demonstrations. At the "ngac>" prompt a predefined procedure (e.g. named "myproc") can be run with the command `proc(myproc).` It can be run with verbose output with the command `proc(myproc,verbose).` It can be made to prompt and wait for user instruction to proceed (empty line input) with the command `proc(myproc,step).`

It is instructive to read the file procs.pl that defines the predefined procedures. The procedures utilise the same commands available at the command prompt. The user may define additional procedures in the procs.pl file for subsequent execution as above.

A sequence of 'ngac' commands can also be stored in a file, in which case it is referred to as a *script*. Scripts may be run with a `script` command, analogous to the `proc` command, with the script file name substituted for the stored procedure name. `verbose` and `step` are valid options also for the execution of scripts.

2.1.3 Implementation

The implementation of the 'ngac' policy tool is comprised of the following Prolog modules:

- ngac.pl – top level module of 'ngac' policy tool; entry point and initialisation
- param.pl – global parameters
- command.pl – command interpreter and definition of the 'ngac' commands
- common.pl – simple predicates that may be used anywhere
- pio.pl – input / output of various policy representations
- policies.pl – example policies used for built-in self-test
- test.pl – testing framework for self-test and regression tests
- procs.pl – stored built-in 'ngac' command procedures
- pmcmd.pl – PM RI command representations and conversions
- spld.pl – security policy language definitions

2.2 'NGAC-SERVER' LIGHTWEIGHT POLICY SERVER

The 'ngac-server' embodies the following elements of the NGAC Functional Architecture: Policy Decision Point (PDP), Policy Access/Administration Point (PAP), and the Policy Information Point (PIP) for policies active at run time.

The policy server may be initiated from within the 'ngac' tool by issuing the command `server( <port> ).` at the tool's command prompt "ngac>". It may also be initiated from a compiled file at the operating system's command prompt.

The 'ngac-server' currently provides two external interfaces, both implemented as RESTful APIs:

- Policy Query Interface – used by a Policy Enforcement Point to query whether a given access should be permitted under the current policy.
- Policy Administration Interface – used by a privileged "shell" or "portal" system program to load and unload policies, combine policies, select policies, etc.

Each of these interfaces will now be described in further detail.

2.2.1 Policy Query Interface (PQI)

A relatively simple interface, in the form of RESTful APIs, constitutes the Policy Query Interface.

This interface is used by a Policy Enforcement Point to determine whether a client-requested operation is supported by the associated user's permissions on the requested object under a particular policy, and if the operation is permitted where may the object be accessed through an appropriate Resource Access Point (RAP).

pqapi/access – test for access permission under current policy

Parameters

- user = <user identifier>
- ar = <access right>
- object = <object identifier>

Returns

- "permit" or "deny" based on the current policy
- "no current policy" if the server does not have a current policy set

Effects

- none

pqapi/getobjectinfo – get object metadata

Parameters

- object = <object identifier>

Returns

- "object=<obj id>,oclass=<obj class>,inh=<t/f>,host=<host>, path=<path>,basetype=<btype>,basename=bname>"

Effects

- none

An active session identifier may be used as an alternative to a user identifier in an access query made to the Policy Query Interface.

2.2.2 Policy Administration Interface (PAI)

The Policy Administration Interface is now a separate interface (different to the EP). As in the EP, policy administration may still be done through the policy tool's command line interface, but in the FP it is best done through the server's RESTful Policy Administration API.

In keeping with the principle of least privilege, the policy administration functions should not be made available to the normal object PEPs. Rather the Policy Administration API should be accessible only to an administratively authorised user through the policy administration tool, and some functions such as setpol/getpol should be accessible only to the "shell" program that executes the NGAC client application. In this way, the "shell" that controls execution of the application would also determine the user/session and policy under which the application should execute. Putting the policy administration APIs in a different location will facilitate limiting their accessibility. Further, the policy administration API now requires pre-authentication and the presentation of a token with each API call.

The enhanced server offers the following APIs as the Policy Administration Interface. A "failure" response is typically preceded by a string indicating the reason for the failure. Implicit in each of the following APIs is an additional parameter, **token**, which is an arbitrary string. The token provided in the call must match the token provided to the server when it was started.

paapi/getpol – get current policy being used for policy queries

Parameters
- none

Returns
- <policy identifier> or “failure”

Effects
- none

paapi/setpol – set current policy to be used for policy queries

Parameters
- policy = <policy identifier>

Returns
- “success” or “failure”
- “unknown policy” if the named policy is not known to the server

Effects
- sets the server’s current policy to the named policy

paapi/add – add an element to the current policy

Parameters
- policy = <policy identifier>
- policyelement = <policy element> only user, object, and assignment elements as defined in the declarative policy language; restriction: only user to user attribute and object to object attribute assignments may be added. Elements referred to by an assignment must be added before adding an assignment that refers to them.

Returns
- “success” or “failure”

Effects
- The named policy is augmented with the provided policy element

paapi/delete – delete an element from the current policy

Parameters
- policy = <policy identifier>
- policyelement = <policy element> permits only user, object, and assignment elements as defined in the declarative policy language; restriction: only user-to-user-attribute and object-to-object-attribute assignments may be deleted. Assignments must be deleted before the elements to which they refer.

Returns
- “success” or “failure”

Effects
- The specified policy element is deleted from the named policy

paapi/load – load a policy file into the server

Parameters
- policyfile = <policy file name>

Returns
- “success” or “failure”

Effects
- stores the loaded policy in the server
- does NOT set the server’s current policy to the loaded policy

paapi/combinepol – combine policies to form new policy

Parameters
- policy1 = <first policy name>
- policy2 = <second policy name>

- combined = <combined policy name>

Returns

- "success" or "failure"
- "error combining policies" if the combine operation fails for any reason

Effects

- the new combined policy is stored on the server

paapi/unload – unload a policy from the server

Parameters

- policy = <policy name>

Returns

- "success" or "failure"
- "unknown policy" if the named policy is not known to the server

Effects

- the named policy is unloaded from the server
- sets the server's current policy to "none" if the unloaded policy is the current policy

paapi/initsession – initiate a session for user on the server

Parameters

- session = <session identifier>
- user = <user identifier>

Returns

- "success" or "failure"
- "session already registered" if already known to the server

Effects

- the new session and user is stored

paapi/endsession – end a session on the server

Parameters

- session = <session identifier>

Returns

- "success" or "failure"
- "session unknown" if not known to the server

Effects

- the identified session is deleted from the server

2.2.3 Policy server command line arguments

When a compiled version of the policy tool or the policy server is started from the command line, several command line options (and synonyms) are recognized.

- `--token`, `-t` <admintoken> use the token make authenticated requests to the paapi
- `--deny`, `-d` respond to all access requests with `deny`
- `-- permit --grant -g` respond to all access requests with `grant`
- `--import --policy --load -i -l` <policyfile> import the policy file on startup

- `--port --portnumber --pqport -p` <portnumber> server should listen on specified port number
- `--selftest -s` run self tests on startup
- `--verbose -v` show all messages

The policy administration functions should not be made available to the normal object PEPs. Rather the Policy Administration API should be accessible only to an administratively authorised user through the policy administration tool or a process with the same authorisation, and some functions such as ***setpol***/***getpol*** should be accessible only to the "shell" program that executes the NGAC client application. In this way, the "shell" that controls execution of the application would also determine the user/session and policy under which the application should execute.

2.2.4 Dynamic policy change

However, it does support *dynamic total policy change*: the ability to load new policies, to form new policies composed of already loaded policies, and to select from among the loaded or composed policies that policy which is to serve as the policy used to make policy decisions. A policy selection remains in effect until a subsequent policy selection. The server retains all of the loaded and composed policies for the duration of its execution. In addition, the current implementation of the 'ngac-server' offers *limited dynamic selective policy change* after a policy is loaded or formed by combining policies. The ***add*** and ***delete*** APIs provide this capability. Details of the limitations are provided in the description of the APIs.

The current implementation of the PIP is ephemeral. There is no persistence of the policy database except in the original policy file(s) used to initialize the server and the sequence of commands issued to the server to modify policies after loading of policy files.

2.2.5 Policy Composition

The policy server supports two forms of policy composition. The first is achieved with the ***comebinepol*** API. I forms the composition of policies as described in the NGAC literature and examples.

The 'all' policy composition is a distinct form of policy composition. When the policy servers current policy is set to 'all' through the ***setpol*** API, all currently loaded policies are automatically combined for every ***access*** request. The manner in which the polices are combined is as follows:

- Every policy is first qualified to participate in computing the verdict of an ***access*** request. To qualify a policy must be defined to have explicit jurisdiction over both the user and the object specified in the ***access*** request. There must be at least one qualifying policy.

- All qualified policies are queried with the triple (user, access right, object) specified in the ***access*** request. If any qualified policy returns 'deny' then the ***access*** request returns 'deny'.

Sets of policies to be combined according to the 'all' policy composition should be designed with the foregoing runtime semantics taken into consideration.

2.2.6 Implementation

The implementation of the 'ngac-server' lightweight server is comprised of the following Prolog modules:

- server.pl – HTTP server for the policy query API and policy admin API
- sessions.pl – registration of session identifiers and associated users to enable sessions identifiers to be used in place of the user in an ***access*** request for the life of the session.

2.3 POLICY ENFORCEMENT POINT (PEP) AND RESOURCE ACCESS POINT (RAP)

Application developers whose applications are to use resources controlled by the NGAC implementation must provide PEPs and RAPs for new resource kinds to create the resource access path as shown in Figure 1.

2.3.1 PEP and RAP Templates

Templates for PEP and RAP are included that provide examples of invoking the Policy Decision Point (NGAC server) and Resource Access Points by Policy Enforcement Points.

The PEP template exhibits the definition of a server for a RESTful Policy Enforcement Interface consisting of the APIs:

- peapi/getLastError – get the error code corresponding to the last error in the API
- peapi/getObject – get an entire object (including opening/closing)
- peapi/putObject – put an entire object (including opening/closing)
- peapi/openObject – open an object for incremental reading/writing
- peapi/readObject – read from an open object
- peapi/writeObject – write to an open object
- peapi/closeObject – close an open object

The RAP template is a example of a RAP for ordinary OS files, and currently implements file_open, file_close and file_read.

2.3.2 Implementation

The implementation of the PEP and RAP templates, as well as a stub for a future unified node manager (if needed), is comprised of the following Prolog modules:

- pep.pl – defines an example PEP server for a policy enforcement interface
- rap.pl – defines a simple example of a RAP for ordinary files
- nodemgr.pl – stub for a unified node manager pulling together PEP, PDP and RAP

2.4 DECLARATIVE POLICY LANGUAGE

We have taken steps toward a lightweight implementation of the NGAC standard. For such a lightweight implementation of NGAC we have developed alternative policy representations, including an alternative representation of the reference implementation's imperative language and our own declarative language. We envision future development of the declarative language to include additional features to facilitate the specification of complex policies for enterprise-wide use and dynamic policy change.

The declarative representation of our 'ngac' policy tool is used to directly declare the entities of a policy and the relations among them. The declarative language is an attractive alternative to using the imperative language as it is much more natural and intuitive. The 'ngac' policy tool uses the declarative representation to build an internal database that represents the policy graph and it includes logic to query the database and to compute access decisions.

The 'ngac' tool may also be used to generate from the declarative form of a policy the equivalent original imperative form for use with the PM Server. After dynamic policy change has been implemented this capability will be modified to generate calls to the RESTful APIs of our lightweight server to effect changes to policies currently loaded into the server.

2.4.1 Declarative Policy Language Definition

A declarative policy specification is of the form:

policy(<policy name>, <policy root>, <policy elements>).

where,

<policy name> is an identifier for the policy definition

<policy root> is an identifier for the policy class defined by this definition

<policy elements> is a list [*<element>, ... , <element>*]

where each *<element>* is one of:

user(<user identifier>)

user_attribute(<user attribute identifier>)

object_class(<object class identifier>, <operations>)

object(<object identifier>)

object(<object identifier>, <object class identifier>, <inh>, <host name>, <path name>, <base node type>, <base node name>)

object_attribute(<object attribute identifier>)

policy_class(<policy class identifier>)

composed_policy(<new policy name>, <policy name1>, <policy name2>)

operation(<operation identifier>)

assign(<entity identifier>, <entity identifier>)

associate(<user attribute id>, <operations>, <object attribute id>)

where *<operations>* is a list:

[*<operation identifier>, ... , <operation identifier>*]

connector('PM')

The initial character of all identifiers must be a lower-case letter or the identifier must be quoted with single quotes, e.g. *smith* or *'Smith'* (identifiers are case sensitive so these examples are distinct). Quoting of an identifier that starts with a lower-case letter is optional, e.g. *smith* and *'smith'* are not distinct.

Additionally:

< inh > can be **yes** or **no**.

< host name> contains the name of the host where the corresponding file system object resides.

< path name> is the complete path name of the corresponding file system object.

2.4.2 Example policy in declarative representation

Earlier we presented two graphical policy examples. These examples are a good resource because they are already developed and well explained in the NGAC and PM documents. Figure 2 presents example (a) in the declarative representation.

Accompanying the declarative policy is a run of the 'ngac' policy tool that interrogates the policy with queries such as **`access('Policy (a)', (u1,r,o1))`**, highlighted in Figure 2. Inset in the figure is the graphical representation highlighted with the path in the graph that enables the **`permit`** response.

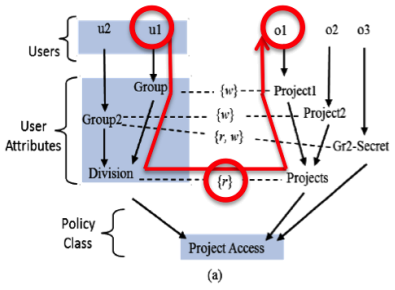
The 'ngac' tool has also been implemented with a built-in test framework that permits a customizable set of test cases to be easily integrated and run as a regression test suite with a single command. The test cases include of both access queries that correspond to the Policy Query Interface of Figure 1 and other testing queries that expose intermediate internal results of policy calculations for diagnostic use as can also be seen in Figure 2.

```
ngac> proc(testA,verbose).
> newpol('Policy (a)').
> los('Policy (a)').
user(u1)
user(u2)
object(o1)
object(o2)
object(o3)
user_attribute(Division)
user_attribute(Group1)
user_attribute(Group2)
object_attribute(Gr2-Secret)
object_attribute(Project1)
object_attribute(Project2)
object_attribute(Projects)
policy_class(Project Access)
assign(u1,Group1)
assign(u2,Group2)
assign(Group1,Division)
assign(Group2,Division)
assign(o1,Project1)
assign(o2,Project2)
assign(o3,Gr2-Secret)
assign(Project1,Projects)
assign(Project2,Projects)
assign(Division,Project Access)
assign(Projects,Project Access)
assign(Gr2-Secret,Project Access)
associate(Group1,[w],Project1)
associate(Group2,[w],Project2)
associate(Group2,[r,w],Gr2-Secret)
associate(Division,[r],Projects)
Connector(PM)
```

```
> dps('Policy (a)').
Group1,r,Project1
Group1,r,Project2
Group1,r,o1
Group1,r,o2
Group2,r,Project1
Group2,r,Project2
Group2,r,o1
Group2,r,o2
u1,r,Project1
u1,r,Project2
u1,r,o1
u1,r,o2
u1,w,o1
u2,r,Project1
u2,r,Project2
u2,r,o1
u2,r,o2
u2,r,o3
u2,w,o2
u2,w,o3

> aoa(u1).
Project1
Project2
Projects
o1
o2

> minaoa(u1).
Projects
```

```
> userlos('Policy (a)',u1).
V= PM
 Project Access
 o1
 o2
 Project1
 Project2
 Projects
E= Project Access,PM
 Projects,Project Access
 o1,Project1
 o2,Project2
 Project1,Projects
 Project2,Projects

> aoa(u2).
Gr2-Secret
Project1
Project2
Projects
o1
o2
o3

> minaoa(u2).
Gr2-Secret
Projects
```

```
> userlos('Policy (a)',u2).
V= PM
 Project Access
 o1
 o2
 o3
 Project1
 Project2
 Gr2-Secret
 Projects
E= Project Access,PM
 Gr2-Secret,Project Access
 Projects,Project Access
 o1,Project1
 o2,Project2
 o3,Gr2-Secret
 Project1,Projects
 Project2,Projects
```




```
ngac> proc(queryA,verbose).
> access('Policy (a)', (u1,r,o1)).
permit
> access('Policy (a)', (u1,w,o1)).
permit
> access('Policy (a)', (u1,r,o2)).
permit
> access('Policy (a)', (u1,w,o2)).
deny
> access('Policy (a)', (u1,r,o3)).
deny
> access('Policy (a)', (u1,w,o3)).
deny
> access('Policy (a)', (u2,r,o1)).
permit
> access('Policy (a)', (u2,w,o1)).
deny
> access('Policy (a)', (u2,r,o2)).
permit
> access('Policy (a)', (u2,w,o2)).
permit
> access('Policy (a)', (u2,r,o3)).
permit
> access('Policy (a)', (u2,w,o3)).
permit
ngac>
```

Figure 2: Example policy (a) expressed in the declarative representation

2.5 IMPLEMENTED FP FUNCTIONALITIES

The functionalities represented in the Full Prototype with respect to the Requirements are summarized in Table 1. It should be borne in mind that D5.2 Early Specification of SPT Framework, Sections 2 and 3, has already presented rationale for the approach to

the SPT Framework and scope of those functional building blocks identified by the IISF that will be implemented as part of the SAFIRE effort. We identified a two-pronged approach, implementing NGAC to make inroads on the problem of system policy specification and configuration, while delegating mundane (and solved) security issues to features already provided by a combination of the operating environment and chosen third party security products. The mapping from the IISF to the Abstract Platform is intended to identify the implementation of those building blocks in the actual deployment (or identify the assumptions that should constrain a deferred deployment).

Abbreviations used in Table 1 are: OE – Operating Environment (OE = OS + HW), 3PSM – Third Party Security Mechanisms, AP – Abstract Platform (AP = OE + 3PSM), EP – Early Prototype, FP – Final Prototype, UC – Use Case, App – Application, SPTM – Security, Privacy and Trust Methodology/Mapping, Impl – Implemented.

The Abstract Platform is "implemented" by the final mapping of the necessary IISF functional building blocks to the elements of a specific deployment on a specific OE with specific selection of 3PSM. This is accomplished according to guidelines provided by the methodology and subject to a risk analysis of the specific environment.

Table 1: Overview of implemented functionality

No.	Requirement	Overall Priority	Status
U94	Ensures data integrity	SHALL	Delegated to the AP
U95	Prevents unauthorized access	SHALL	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation TBI
U96	Provides secure access to generated knowledge in the cloud	SHALL	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation TBI
U98	Provides at least the same level of security as afforded by the operating environment	SHALL	Level commensurate with that afforded by OE, cannot provide greater
U99	Support for different user roles for secure access control	SHALL	Impl by NGAC with AP support
U100	Supports established authentication capabilities (hardware / passwords…)	SHALL	Delegated to the AP
U101	Provides data confidentiality management facilities	SHALL	Impl by NGAC EP with AP support, subject PEP/RAPs and UC App adaptation TBI
U102	Provides secure connectivity from machines to cloud	SHALL	Delegated to the AP
U103	Provides support for VPN connectivity	SHOULD	Delegated to the AP
U104	Provides encryption of data at rest locally	SHALL	Delegated to the AP
U105	Provides encryption of data at rest in the cloud	SHALL	Delegated to the AP
U106	Provides encryption of data during	SHALL	Delegated to the AP

	transport		
U107	Supports the establishment of rules/policies of trustworthiness (safety, privacy, security, reliability, resilience)	SHALL	Impl by NGAC FP and the SPTM
U108	Allows sharing of analysis results with designated people	SHOULD	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation TBI
U109	Allows collaboration of designated groups for building a knowledge base	SHOULD	Impl by NGAC FP with AP support, subject PEP/RAPs and UC App adaptation TBI
U110	Authenticates who/what is sending data	SHOULD	Delegated to the AP

3. INTEGRATION WITH OTHER MODULES

A single NGAC server instance can support multiple policies simultaneously. Alternatively, multiple instances of the NGAC server can be run to server clients running under distinct policies in a large distributed system. Several applications of NGAC, in addition to the primary mission of endpoint data protection access control, may be possible within the IISF catalogue of security functional building blocks. For now, we focus on NGAC-aware applications and SAFIRE infrastructure.

3.1 NGAC-AWARE APPLICATIONS

An NGAC-aware application is one that has been ported to access its critical controlled resources through NGAC's policy enforcement interface. This is accomplished by modifying the application to replace direct resource access methods with calls to an NGAC PEP.

Our approach to the NGAC Functional Architecture has been to "unbundle" the PEPs and RAPs from the NGAC infrastructure and provide a policy server API that can be called by custom PEPs. This approach places the adoption of NGAC within the application developer's domain and control. The application and the PEPs and RAPs for the resources it accesses are all developed by the application developer according to a simple architectural pattern. A call is made by the PEP to the PDP to determine whether a particular access (consisting of: user, access right, object) is permitted under the policy before calling the appropriate RAP to access the resource. This call to the PDP may even be stubbed out (returning 'grant') during development or in the absence of an operating server so that the developer's work is not hindered. The developer is encouraged to develop RAPs that use the same resources and resource servers, hence the same Resource Access Interface that the application would have used in an unmediated scenario.

3.2 POLICY ENFORCEMENT POINT (PEP) AND RESOURCE ACCESS POINT (RAP) ARCHITECTURAL PATTERN

Client applications (APP) of the NGAC-managed object space must request access to objects through the Policy Enforcement Interface. The developer of the client must

create a PEP and a RAP for the objects required by the application, if such PEP and RAP do not already exist for objects of that kind. The architectural pattern to be used is very straightforward but should be followed faithfully if the integrity of the access control is to be maintained. (See Figure 1.)

The architectural pattern is designed to achieve this, through two essential features. First, the PEP must be a distinct process from that of the application, able to be executed under an identity distinct from that of the user executing the application. The same is true for the RAP. Second, the PEP and the RAP are very simple in their purpose and function, and thus are relatively easy to assure by inspection of their code and of the permissions given to their executables.

The PEPs and the RAPs, as well as the NGAC server itself, execute as a distinct 'ngac-user' identity. The intrinsic access control mechanisms of the underlying platform operating system (OS) are used to segregate all NGAC-managed objects and to provide access to these objects exclusively to processes executing on behalf of the 'ngac-user' identity.

Finally, in a deployed industrial scenario, where there are enterprise-critical resources and untrusted users and agents, the communications among the APP, PEP, PDP, and RAP should be performed over encrypted and authenticated sockets. This step requires an additional layer of key generation, distribution, and management that is well understood in practice but not done in the prototype.

3.2.1 PEP Policy Enforcement Interface (peapi)

A relatively simple interface, in the form of RESTful APIs, constitutes the Policy Enforcement Interface.

peapi/getobject – return object content (read)

Parameters

- object = <object identifier>

Returns

- "success" or "failure"
- <object data>

peapi/putobject – set object content (write)

Parameters

- object = <object identifier>
- content = <object data>

Returns

- "success" or "failure"

3.2.2 RAP Resource Access Interface (raapi)

Resource Access Points embody access methods that would likely be used directly in the client application if it were to access the resource without mediation by NGAC. The RAPs may be specific to a kind of object and may be very similar to a code fragment that would have been used in the application under such a scenario. Thus, the details of the Resource Access Interface may be chosen by the application developer, and

encapsulated in a small distinct process. The PEP determines the appropriate RAP to call after asking the PDP for a policy verdict and getting the object's metadata from the PIP through a PQI call (***getobjectinfo***).

4. INSTALLATION AND OPERATION

4.1 INSTALLING AND RUNNING THE 'NGAC' POLICY TOOL

The ngac policy tool is implemented in Prolog and requires the SWI Prolog environment to run. The ngac tool can be provided as a set of Prolog source files and/or as an "executable" that has the Prolog runtime environment already bundled in. This executable is made by the shell script mkngac, located with the source files that must be run in an environment that has SWI Prolog installed.

4.1.1 Install SWI-Prolog

SWI Prolog is available for several operating environments, including Mac, Windows, and Linux. See http://www.swi-prolog.org.

4.1.2 Install the 'ngac' source files and/or executable

The current version of the ngac tool consists of a directory tree including source files and example files. The distribution is provided as a zip file of this directory tree.

4.1.3 Initiate the 'ngac' policy tool

If a ready made executable 'ngac' has been provided it may be executed directly from a command shell prompt. If you do this skip down to "Now you should see …" below.

Otherwise, in the source directory ngac-server-2018-06 start SWI-Prolog from a command shell prompt using the name of the SWI-Prolog executable (usually 'swipl', 'swi-pl', or something similar, depending on how it was installed).

After printing a short banner SWI-Prolog will display its prompt "?- ".

At the Prolog prompt enter "[ngac]." (not the quotes)

Prolog will compile the code and print "true."

Execute the code by entering at the Prolog prompt "ngac."

Now you should see the 'ngac' prompt "ngac> '

4.1.4 Test the installed 'ngac' tool

The ngac tool has some self-tests built in. These should be run to ensure that everything is working correctly. Follow the instructions in the preceding section to run the ngac tool. Start it with the Prolog prompt command "ngac(self_test)." This will run some built-in self tests when it starts. To not run the self-tests simply start the tool with the Prolog prompt command "ngac."

The self tests can also be run by starting 'ngac' normally and entering at the 'ngac' prompt "selftest."

Procedures make up of 'ngac' commands may be predefined in the procs.pl file. Look at the ones there and try them by entering the ngac command "proc(ProcName).", where ProcName is the name of one of the procedures defined in procs.pl.

4.1.5 Running the examples

There are several examples included with the sources of the ngac Policy Tool. These include examples described in documents and PowerPoint slide decks used to introduce the NGAC concepts.

There are predefined procedures ("procs") that run the examples. At the ngac> prompt a predefined procedure (e.g. named "myproc") can be run with the command `proc(myproc).` It can be run with verbose output with the command `proc(myproc,verbose).`

It is instructive to read the file procs.pl that defines the predefined procedures. The procedures consist of the same commands available at the command prompt. The user may define additional procedures in the procs.pl file for subsequent execution as above.

4.2 INSTALLING AND RUNNING THE 'NGAC-SERVER'

The ngac server is implemented in Prolog and requires the SWI-Prolog environment to run. The server can be provided as a set of Prolog source files and/or as an "executable" that has the Prolog runtime environment already bundled in. This executable is made by the shell script mkngac, located with the source files that must be run in an environment that has SWI Prolog installed.

4.2.1 Install SWI-Prolog

SWI Prolog is available for several operating environments, including Mac, Windows, and Linux. See http://www.swi-prolog.org.

4.2.2 Install the 'ngac' server source files and/or executable

The current version of the ngac server consists of a directory tree including source files and example files. The distribution is provided as a zip file of this directory tree.

4.2.3 Initiating the 'ngac-server'

In the EP, the ngac server is started form the 'ngac' policy tool. In the FP the ngac server may be started with a compiled executable. This is preferable since it allows the command line options to be specified.

If you do want to start the server from the policy tool follow the instructions above to get 'ngac' running. After starting 'ngac' it offers the prompt "ngac>". There are a set of basic commands available in the normal mode (admin) and an extended set of commands for use by a developer in development mode (advanced). Entering the command "help" will list the available commands in the current mode. If you want to load any policy files, do it now with the 'ngac' command "import(policy(PolicyFileName)).", where PolicyFileName is the name of a .pl file relative to the execution directory. You can also combine policies with the 'ngac'

"compose" command. When you have the desired policies loaded and composed, start the server from the 'ngac' tool using the command "server(PortNumber).", where PortNumber is an unused TCP port. The server will be started and will be listening to that port for calls to its RESTful API. A server started in this way will expect the default admin token (the string "admin_token", without the quotes) in the policy administration API calls.

4.2.4 Test the installed 'ngac-server'

There is a shell script of curl commands included with the source (servercurltest.sh) in the TEST subdirectory. This script can be run to send a sequence of requests to the server to test for known correct answers.

5. NGAC CUSTOMISATION

5.1 CURRENT CUSTOMISATIONS AND EXTENSIONS

These are customisations and extensions represented in our implementation. These extensions in some cases are ones that have been done to a previous version of our implementation, and in some cases are extensions to the NGAC Standard that have already been done or tentatively planned to be done.

5.1.1 Adaptation and extensions to NGAC for SAFIRE

SAFIRE SPT Framework seeks to make advancements by exploring coherent system-wide security policy and enforcement in IIoT systems. Thus the SAFIRE Security Framework must provide an expressive, dynamic, and comprehensible security policy description and enforcement vehicle. The base requirements are met by the policy modelling and enforcement framework expressed in the NGAC Functional Architecture and related standards.

We intend to demonstrate how the application of NGAC can provide a unifying approach to policy definition and access control in the dynamic FoF IIoT environment that is the result of reconfiguration and optimisation as in SAFIRE. To do so we will have to make NGAC more usable, deployable in more diverse environments than its past reference implementations, make it to be more easily extensible for new kinds of protected objects, and make it more supportive of large complex policies and policy composition.

We have described our own version of the NGAC functional architecture with “unbundled” PEP and RAP as shown in Figure 1. This architecture enables a more extensible implementation of NGAC by easing the addition of new protected object kinds. Another feature of our version of the architecture is a lightweight Policy Server that places fewer demands on the operating environment and thus is intended to be more portable.

Other extensions of our current implementation include:

- Extensions to the declarative policy language
- Support for policy composition
- Unbundled PEPs and RAPs for new object classes
- Constrained dynamic policy change
- Full Policy Query and Policy Administration interfaces

Declarative Policy Language
We have previously developed a simple declarative policy specification. Our expressed intention was to implement additional features in the language, including:

- *Prohibitions*, a construct to explicitly specify accesses that are to be denied (in addition to *default deny*), is a feature of the NGAC policy framework not previously implemented in our declarative language. Prohibitions were not implemented as we have not found a need for them in the use cases that we have considered.

- Ability to define new object classes – Currently, object classes are arbitrary identifiers that are not checked. This has been done.

- Ability to define specific operations that correspond to an object class – This will permit more precise type checking of object operations in policy rules. This has been done.

- Explicit support for policy composition in the language – Previously, policy combination was a command in the 'ngac' tool but not part of the language. The needed features are discussed more in the following section. Composition has been fully implemented in the policy server as well as in the policy tool. This has been provided as an alternative to the composition declaration in the language.

The definition of the declarative policy specification language given in APPENDIX A – has been extended to include the new language features in these extensions.

Policy Composition

For large heterogeneous IIoT systems there needs to be a representation in the policy language for the composition of policies and support for various forms of composition. The previous implementation of the 'ngac' policy tool did not fully support combined policies such as the example (Figure 5) in the APPENDIX. The implementation has now been extended to correctly handle such a simple examples. However, we anticipate a need to express policy composition *within the language* not just as a command outside of the language. We have extended the declarative language to include declarative policy composition and the implemented language processor accepts this extension. The processing to internally generate the declared composed policy has not been completed in the FP although much of the needed machinery is in place. The need has been met by full implementation of composition through the server policy administration API.

Unbundled PEPs and RAPs for new Object Classes

Application developers need to be able to adapt to NGAC without waiting for further NGAC development before their needed resources can be available under NGAC authority. The developer already knows the resource and how it is to be accessed by the application. It should not be necessary to modify the core NGAC implementation for every such example. Instead, the developer should be able to construct all of the components of the object access path and test the application without involvement of the NGAC server if necessary. The necessary components are small trusted components that consist primarily of code that probably exists already in some form with the application. The developer must extract relatively small bits of code from the application and place them into templates for Policy Enforcement Points (PEP) and

Resource Access Points (RAP). The Policy Enforcement Interfaces and Resource Access Interfaces are thus under the control of the developer.

We have developed examples and templates for PEPs and RAPs that application developers can use to more easily develop PEPs and RAPs, enabling them to add new kinds of protected resources.

5.2 EXTENSIONS IN FULL PROTOTYPE

5.2.1 Importing Policies to the Server

A rich API has been added to the policy server for policy administration, including loading, unloading, and combining of policies.

5.2.2 Modifying Policy at Runtime

In addition to loading policies the server has been extended with ***add*** and ***delete*** APIs to modify loaded policies by adding or deleting individual policy elements and assignment relations. This is a limited form of dynamic policy modification that includes:

- Add user
- Add user attribute that is not part of an association (non-associated)
- Add assignment of a new or existing user or user attribute
- Add object
- Add non-associated object attribute
- Add assignment of new or existing object or object attribute
- Delete user and assignment of user to user attribute(s)
- Delete object and assignment of object to object attribute(s)
- Delete empty non-associated user attributes or object attributes

Note that this entails restrictions. There is no addition or deletion of associations and no deletion of associated (user or object) attributes.

Policy change is achieved through a sequence of calls to the ***add*** and ***delete*** APIs of the Policy Administration Interface. Appropriate checks are made when these calls are made to keep the policy structure in a consistent state. Assignments must be deleted before the entities assigned can be deleted. Conversely, entities must be added before assignments can be made.

5.2.3 Policy Composition

The SAFIRE implementation of the policy server has corrected previous deficiencies in policy composition by modifying the internal data structures and algorithms. This version also includes a new distinct form of composition of ‘all’ loaded policies.

5.2.4 Persistence of the Server Policy Database

Currently policies may be loaded into the server and limited forms of change made. We considered implementation of persistence of the PIP but have not implemented persistence in the FP. We currently have a constrained form of policy change and it is not yet clear that starting from a previous (possibly uncertain) state, rather than a deterministically reproducible state, is better. By recording the sequence of changes to a known initial policy state (from a policy file) the state after changes can be reliably reproduced. We will reconsider persistence of the PIP if any of our use cases require it.

6. SOFTWARE TOOLS

The implementation of the 'ngac' policy tool and the lightweight 'ngac-server' are intended to be simple and portable, with minimal external dependencies. Motivation for its development and the specific objectives are described elsewhere. The NGAG reference implementations have been very heavyweight with many external dependencies on languages, libraries and tools.

Our NGAC software is implemented in the Prolog language, which is well suited to representation and computation over access control policies in the NGAC framework. The Prolog implementation we use, SWI-Prolog, includes a visual editor and graphical tracer and a built-in make facility for rapid iterative development.

The only tools necessary for this implementation are SWI-Prolog and the libraries included in its release; we are currently using version 7.6.4. The release includes a version of Emacs with a Prolog display profile, though an editor of the developer's choosing may be configured for invocation instead when editing Prolog source files.

7. CONCLUSIONS AND PLANS

We have succeeded in implementing a version of the NGAC standard that computes the policy calculations for an arbitrary policy or combinations of policies. Existing examples of NGAC policies, and examples added to exhibit newly implemented features, are used as built-in self-tests and regression tests to confirm that the tool computes the known answers and that changes made during development do not cause the implementation to regress from previously achieved correct operation.

Several extensions to the EP were developed to create the FP. These include those capabilities needed to fulfil the contemplated application as a mechanism for endpoint access control in the present use cases.

We intend to continue to develop and enhance our NGAC implementation to meet new requirements that we encounter as we pursue our exploitation plans.

As has been demonstrated in our mapping of the IISF functional building blocks to the NGAC approach, there are numerous future potential options for using NGAC in multiple functional roles within deployments of a SAFIRE-enabled FoF system. When an NGAC-based approach seems that it would be beneficial, adding features to the present implementation and already planned extensions will be undertaken to further expand the capabilities and applicability of the implementation.

8. REFERENCES

[FGJ15] David Ferraiolo, Serban Gavrila, and Wayne Jansen. Policy Machine: Features, Ar- chitecture, and Specification. National Institute of Standards and Technology, October 2015. NIST Internal Report 7987 Revision 1.

[G+] Gavrila et al. Policy machine source. https://github.com/PM-Master.

[Gav07] Serban I. Gavrila. The Policy Machine: User Guide, January 2007.

[Int15] InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control—Generic Operations & Abstract Data Structures, May 2015. INCITS Project CS1/2195-D, NGAC-GOADS, Revision 1.60 in review, to ap- pear.

[Int16a] InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control—Functional Architecture, February 2016. INCITS Project CS1/2194-D, NGAC-FA, Revision 0.70 in review, to appear.

[Int16b] InterNational Committee for Information Technology Standards, Cyber security technical committee 1. Information technology—Next Generation Access Control— Implementation Requirements, Pro- tocols and API Definitions, February 2016. INCITS Project CS1/2193-D, NGAC-IRPAD, Revision 0.20 in review, to appear.

9. APPENDIX A – NEXT GENERATION ACCESS CONTROL

9.1 NGAC OVERVIEW

The access control policy specification representations to be described are based on the framework provided by the Next Generation Access Control (NGAC) standard:

- Next Generation Access Control Functional Architecture (NGAC-FA) [Int16a]
- Next Generation Access Control Generic Operations and Data Structures (NGAC-GOADS) [Int15], and
- Next Generation Access Control Implementation Requirements, Protocols and API Definitions (NGAC-IRPAD) [Int16b].

Where the standards leave off the "Policy Machine" reference implementation is consulted for details.

We begin with a description of the access control framework. Following this, we describe the representations found in the standards or in the reference implementation, including a graphical representation, the low-level textual representation, and the low-level imperative command representation are distilled from the standards and the reference implementation. The low-level imperative command representation of policy is that of the reference implementation described in [Gav07]. It is capable of dynamic policy changes that can be done under the control of scripts triggered by events.

Also presented are some alternative policy representations that we have developed, including an alternative to the low-level imperative command representation and a novel declarative representation used by our 'ngac' policy tool. The declarative representation is cleaner and more intuitive. We have not yet investigated its extension for dynamic policies though our tool does have scripting capabilities that may be extended and adapted for the purpose.

Examples and figures following are taken from the NGAC standard document and from documents describing the Policy Machine [FGJ15, Gav07] reference implementation [G^+] of the NGAC standard. The examples are recast in the alternative declarative representation.

9.2 NGAC ROLE IN SAFIRE ARCHITECTURE

The IISF highlights the foundational role of security model and policy in its functional viewpoint. Critical aspects of this role are those addressed by NGAC.

NGAC is a novel approach to access control that affords unprecedented flexibility and the ability to represent and enforce arbitrary attribute-based access control policies within a unified framework. NGAC has not to our knowledge been applied previously in industrial manufacturing or any Industrial Internet of Things (IIoT) environment.

NGAC extensions will also be needed to address SAFIRE real-time data requirements for security policy enforcement.

The features and particulars of the NGAC reference implementation, and the published examples, suggest that it has thus far been applied to enterprise IT. However, the scalability and flexibility of NGAC make it a potent tool for addressing the scale, complexities, and security challenges of combined IT and OT as found in IIoT systems.

NGAC provides a framework and mechanisms that have the potential to unify the system-wide access control policies, and provide the ability to understand the net effect of the composed policies of the underlying mechanisms.

There are several extant reference implementations (RI) of NGAC, mostly under the name "Policy Machine" (PM), that have been developed in recent years by the principal authors of the NGAC standards. It has not been an apparent goal of these efforts to provide an industrially deployable implementation that supports heterogeneous environments, or that is readily extensible to new kinds of protected objects. Rather, the re have all been proofs-of-concept, all the more so with the latest versions. Nonetheless, with versions subsequent to the initial PM, the developers seem to have taken into consideration some of our comments and requests made over the past two to three years, with improvements such as independence from Microsoft Windows Server, independence from Windows Active Directory, and independence from LDAP. They have moved to store the policy information first in MySQL and then, as an option, in Neo4j[1], which is better suited to policies in the NGAC framework. As more recent prototypes have emerged, they seem to have focused on narrower aspects of NGAC functionality, rather than a complete system, such as improved algorithms, a Web-service-based NGAC server, RESTful APIs, and deployment in Docker containers, though, as far as we understand, not all backward-compatible with their more complete early PM versions. We continue to monitor the releases of their experiments, as it is clear that many of their recent developments are relevant to our concerns.

NGAC is described in the source documents [FGJ15], [Int15], [Int16a] and [Int16b]. Several versions of reference implementations of NGAC are described in [G+] and [Gav07].

The Open Group has implemented some of our own NGAC-related tools and a simple declarative language to express policies that comply with the NGAC framework. Specifically, a desktop command-line tool called 'ngac' that loads policies expressed in the declarative language and can answer queries such as "access(policy1,(u1,r,o2))", the meaning of which is: "under policy 'policy1', is user 'u1' allowed to read object 'o2'?".

The declarative language is easily read from a graphical representation of the policy, and is more intuitive than the imperative language of the PM RI. The 'ngac' policy tool can generate a translation of the declarative policy in the imperative language for import into the earlier PM servers. The present declarative language does not support the entire NGAC policy framework, lacking prohibitions and obligations. The present

[1] Neo4j is a graph database that is available in community and commercial versions.

specification of the declarative language is given in the text of this document. We anticipate making extensions to this implementation in the future as enumerated in Section 5.1.1.

9.3 NGAC MOTIVATION

According to the NGAC-FA Standard [Int16a]:

> *Next Generation Access Control (NGAC) is reinvention of traditional access control into a form that suits the needs of modern, distributed, interconnected enterprise. The NGAC framework is designed to be scalable, to support a wide range of access control policies, to enforce different types of policies simultaneously, to provide access control services for different types of resources, and remain manageable in the face of change.*

According to the NGAC-FA Standard [Int16a]:

9.4 NGAC POLICY FRAMEWORK

The core constructs of the access control framework are:

- A set of basic elements – representing entities
- A set of containers of different types – to represent characteristics of basic elements
- A set of relations – to represent relationships among basic elements and containers

There is also a set of commands for the creation, deletion and maintenance of basic elements, containers and relations.

The basic elements comprise:

- Users – unique entities that are either humans, trusted programs, or devices
- Processes – system entities that have a reliable user identity and operate in a distinct memory
- Objects – resources to which access is controlled, e.g. files, messages, database records, etc.
- Operations – denote actions performed on elements of policy (either external protected resources or internal resources)
- access rights – enable actions to be performed on elements of policy (either external protected resources or internal resources)

Containers comprise:

- User attributes – defines membership on the basis of an abstract user capability or prop- erty. The members of a user attribute may be users or other user attributes. Membership is transitive.
- Object attributes – defines membership on the basis of an abstract object characteristic or property. Members of an object attribute may be objects or other object attributes. Membership is transitive.

- Policy classes – defines membership related to an access control policy, such as RBAC, MLS. Members of a policy class may be users, user attributes, object, or object attributes. Multiple policy classes may exist simultaneously.

Every user attribute, object attribute, and policy class has a unique identifier. Policies are expressed as configurations of relations of the following four types:

- Assignment – defines membership within containers, involves a pair of policy elements
- Association – defines authorized modes of access, it is a 3-tuple
  < *userattribute, accessrightset, attribute* >
- Prohibition – specifies a privilege exception, it is a 4-tuple (of 3 different kinds described below)
- Obligation – dynamically alters access state, triggered by an event; it is a 3-tuple
  < *user, eventpattern, eventresponse* >

The prohibition relation has three forms:

- <*user, accessrightset, inclusiveattributeset, exclusiveattributeset*>
- <*userattribute, accessrightset, inclusiveattributeset, exclusiveattributeset*>
- <*process, accessrightset, inclusiveattributeset, exclusiveattributeset*>

Events that trigger obligations may include the following information:

- Operation;
- User ID;
- Process ID;
- One of the user attributes of the process performing the operation;
- Policy element ids representing a resource or policy information; or
- One or more attributes of the resource or policy information on which the operation has been performed.

From the four configured relation types above, four types of derived relations can be computed for the purpose of making access control decisions:

- Access control entry – derived from association; < user, accessright >
- Capability – derived from association; < accessright, policyelement >
- Privilege – derived from association; < user, accessright, policyelement >
- Restriction – derived from conjunctive and disjunctive prohibition relations of the same form; restricts process from performing an operation against a policy element, based on process attributes.

10. APPENDIX B – NGAC-BASED SECURITY POLICY REPRESENTATIONS

There are several representations of policies based on the NGAC policy framework.

10.1 GRAPH REPRESENTATION

Policies expressed in this framework are best considered as mathematical graphs. A graphical user interface (GUI) is provided within the PM Admin tool of the NGAC reference implementation. Currently, the implementation of this GUI is limited to provide only views that can be represented as trees, and therefore it is not a particularly good visualization of the policy graph. For the examples presented here a more natural representation as unified directed graphs is used.

Figure 3 illustrates two examples of assignment and association relations as graphs. Figure 3(a) is an access control policy configuration with policy class "Project Access", and Figure 3(b) is a data service configuration with "File Management" as its policy class. On the left side of each graph are users and user attributes, and on the right side are objects and object attributes. Arrows represent assignment relations and dashed lines represent associations. An association may also be thought of as a pair of assignments, the first an assignment of a user attribute to an operation set, and the second an assignment of the operation set to an object attribute. In the association **Division–{r}–Projects**, the policy elements referenced by **Projects** are objects **o1** and **o2**, meaning that users **u1** and **u2** can read objects **o1** and **o2**.

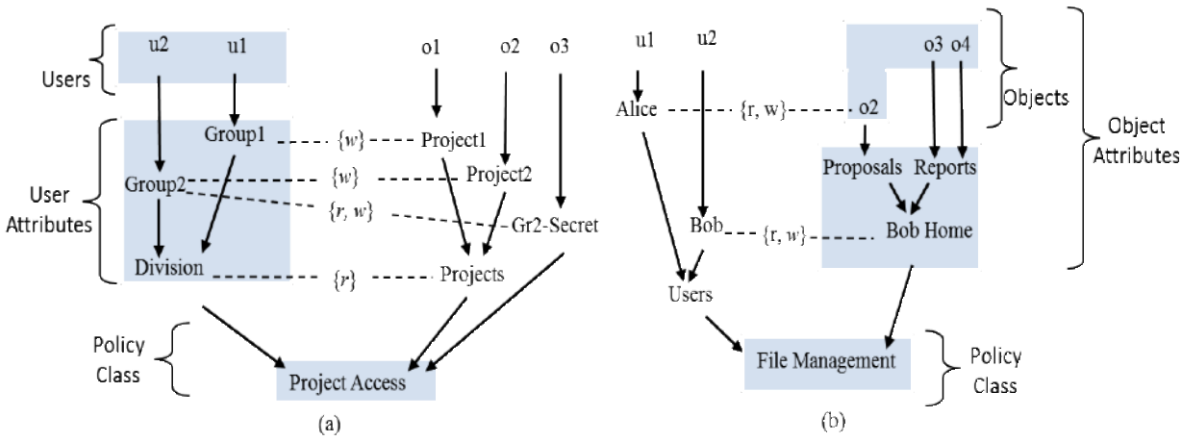


Figure 3: Assignment/Association Graphs

Figure 4 illustrates the independent derived privileges of the separate graphs in Figure 3.

(u1, r, o1), (u1, w, o1), (u1, r, o2), (u2, r, o1), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3)	(u1, r, o2), (u1, w, o2), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3), (u2, r, o4), (u2, w, o4)

Figure 4: Independent derived privileges from Figure 3(a) and (b)

The policy specification technique allows for complex policies to be built up from separate policies or policy fragments. For example the two policies of Figure 3 when combined yield the policy graph illustrated in Figure 5.

The derived privileges of the combined graphs (Figure 5) of Figure 3 are shown in Figure 6.

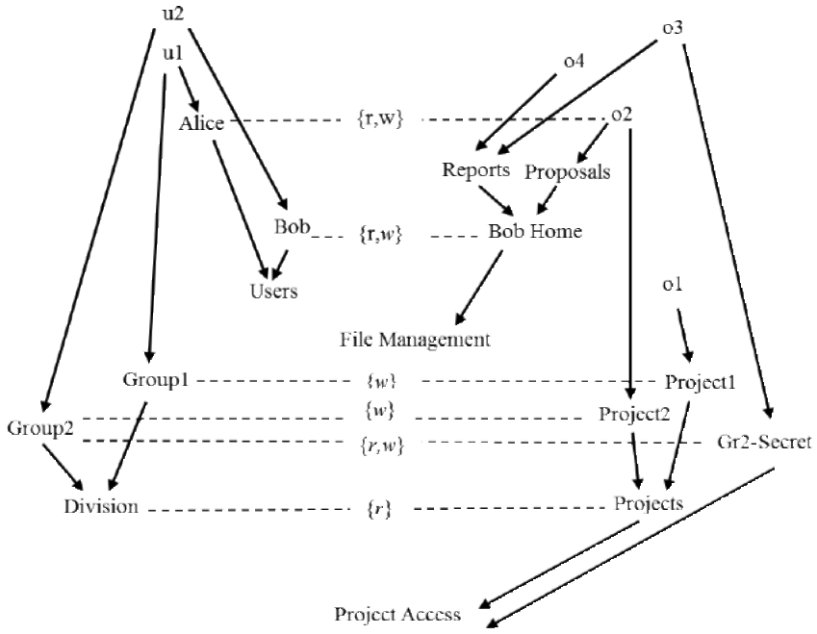


Figure 5: Combined policy graphs of Figure 3

(u1, r, o1), (u1, w, o1), (u1, r, o2), (u2, r, o1), (u2, r, o2), (u2, w, o2), (u2, r, o3), (u2, w, o3), (u2, r, o4), (u2, w, o4)

Figure 6: Derived privileges of the combined graphs of Figure 3

10.2 LOW-LEVEL REPRESENTATIONS

Earlier PM reference implementations used two low-level representations, a low-level textual representation of the policy machine graph, and a low-level imperative command representation that represents primitive actions that the Policy Server can perform to build and manipulate elements of its internal policy model. The Admin Tool in the PM reference implementation directly manipulates the system-wide model maintained in the PM Server by passing it commands in an imperative language.

Live update of the system-wide model can easily disrupt normal operations, and is not a viable practice for policy development and testing. We have implemented a standalone policy tool called 'ngac' for this purpose. The low-level imperative representation can be generated by our 'ngac' policy tool, creating files that can be imported by the PM.