Policy Enforcement for Secure and Trustworthy Data Sharing in Multi-domain Infrastructures

Xin Zhou*
University of Amsterdam
Amsterdam, The Netherlands
x.zhou@uva.nl

Reginald Cushing[†]
University of Amsterdam
Amsterdam, The Netherlands
r.s.cushing@uva.nl

Ralph Koning
University of Amsterdam
Amsterdam, The Netherlands
r.koning@uva.nl

Adam Belloum
University of Amsterdam
Amsterdam, The Netherlands
a.s.z.belloum@uva.nl

Paola Grosso
University of Amsterdam
Amsterdam, The Netherlands
p.grosso@uva.nl

Sander Klous
KPMG
Amsterdam, The Nethrelands
klous.sander@kpmg.nl

Tom van Engers
University of Amsterdam
Amsterdam, The Netherlands
T.M.vanEngers@uva.nl

Cees de Laat
University of Amsterdam
Amsterdam, The Netherlands
delaat@uva.nl

***Abstract*—The push for data sharing and data processing across organisational boundaries creates challenges at many levels of the software stack. Data sharing and processing rely on the participating parties agreeing on the permitted operations and expressing them into actionable contracts and policies. Converting these contracts and policies into an operational infrastructure is still a matter of research and therefore begs the question of how should the software stack and infrastructure look like? What are the main building blocks of such an architecture? In this paper, we investigate the architecture of a multi-domain distributed architecture for policy driven application. The architecture spans components from auditing policies to managing network connections.**

***Index Terms*—multi-domain data sharing, policy enforcement, overlay network, trust distribution.**

I. Introduction

Security of transferring data recently became an important topic because of the value that is created from data. Therefore protecting data from unauthorised parties becomes ever more relevant [20, 7]. To guarantee the safety of data, many policies and regulations have been designed for ruling the collection, the sharing, and the transfer of data. However, when collaborating with different individuals, departments or organisations [11], the collaborators must be able to freely share some information. Especially when the requirement for timeliness is high, for example in emergency management, the effective crisis response relies on efficient information flow among the involved parties. Since polices may prohibit the effective transmission of information [1], they may thus hamper efficient flow of information between the collaborators. This trade-off between protecting data and sharing it among collaborators thus poses a practical problem [16]. However, by sharing data in a system that can enforce the policies, we may find a solution to this problem. Therefore, the main questions we try to answer are: 1) *How can we model policies for use in multi-domain distributed infrastructures as a means?* 2) *How can we operationalise collaborative applications on decentralised infrastructures while maintaining strict policy adherence and tight security controls?*

To answer these questions, we use a real world use-case for emergency management: ArenA. Based on this use-case, we model the policies, and enforce them on a distributed multi-domain infrastructure. The paper is structured as follows: Section II defines the our conceptual model that introduces the manifest, the auditor, and the components to map the policies to executable actions. We then give a short introduction to Jason (Sec. III) by which we realise the policy enforcement. Our use-case and the implementation of its data transmission auditing process is presented in Sec. IV.

After defining the auditing process, we show how this process is integrated in our infrastructure actor architecture in Sec. V. Finally, we discuss our proposed method and implementation in Sec. VI and conclude.

II. Conceptual model

To guarantee the data processing being monitored, queried, and traced, we introduce the concept of **manifest** which contains metadata of the dataset. In addition, to ensure the designed data protection policies can be enforced by the infrastructure, a conceptual model is proposed to translate the policies into conditional statements and into executable actions.

A. Manifest

The manifest contains metadata for each dataset, it describes the included files/datasets, the controller domain, the permitted recipients' domain, the corresponding applied policies, the sender domain, and a timestamp. This manifest will be generated by the data controller or by successfully transferring of the dataset automatically. It serves for the auditors to ensure what policies they should comply to with regard to a certain dataset, also make the data operation can be traceable as it record when and where does the dataset come from. The manifest must be signed using the cryptographic key of the data controller to

[†] and * contributed equally to this paper. This research is funded by the Dutch Science Foundation in Commit2Data program (grant no: 628.001.001).

prevent other parties from tampering with the policies or other metadata.

Table I shows the fundamental structure of the manifest, describes the included metadata, and gives examples of their values.

Item	Value
Datasets	Set of files {Name of the file} Eg: {File_1,File_2}
Controller domain	The domain name of the data controller Eg: Alice
Policies	Set of policies {Name of the policy} Eg:{Policy_1, Policy_2}
Sender domain	The domain name of the data sender Eg: Alice
Recipient domain	The domain name of the recipient Eg: Bob
Timestamp	The timestamp of the manifest generation Eg: 20161206 9:34:10

TABLE I
MANIFEST: METADATA OF DATASETS/FILES

B. Auditor

We will now introduce the concept of "**auditor**" into the data transferring network, who has the operation history log, and is able to check if the request of action is in line with the designed policies to decide whether to authorise the request or not. Only when the request is authorised by all the required auditors it can be executed on the infrastructure. The auditor, thus, guarantees the enforcement of the policies by using the policy components mentioned in Sec. II-C. The natural language policies can be translated into conditional statements and executable actions, so that the auditors can compare, review, and then output the decision to realise automatic or semi-automatic auditing. We chose Jason [2, 17], one of the most solid development environments for a belief-desire-intention (BDI) [12, 14] based multi-agent system(MAS), to model the auditing process. In Sec. IV-B, we elaborate the concrete Jason implementation.

C. Policy components

Policies that apply to a certain dataset are regulated by the data controller who controls the access and usage [13, 9]. These policies are used by the auditors who judge if the requests comply with the policy. When the auditor approves such a plan, it will be signed using a cryptographic key.

Fig. 1 presents the conceptual view of the policy which contain authorisations, obligations, and environmental conditions [15, 10].

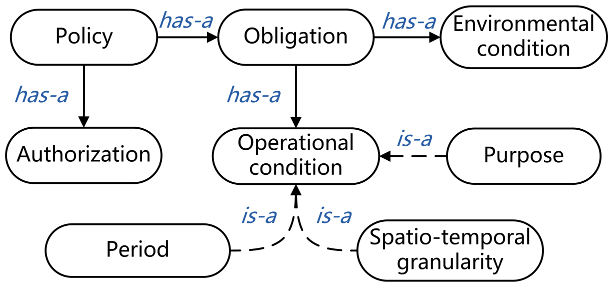


Fig. 1. Conceptual view of the policy. It regulates the required auditors for the "Obligation" in "Authorisation". The "Obligation" can be vary with "Environmental condition". Thus the latter is also specified in the policy. The clarification of purpose, period, and spatio-temporal granularity in the "Operational condition" defines the rights and duties in a fine-grained way.

1) Authorisations: Authorisations are used to indicate the auditors who are going to evaluate the plan/request and check if the contained actions are allowed to be performed. For example, the data controller can appoint $auditor_1$ and $auditor_2$ as authorised auditors for $dataset_1$, which means that all the requests that operate on $dataset_1$ need the signatures given by these two auditors before executing.

2) Obligations: Obligations are used to describe the mandatory requirements to which the subject needs to adhere. For example data controller "Alice" is obligated to transfer "dataset_1" to "Bob".

3) Environment Conditions: The environmental conditions describe the required environmental or system requirements for a certain operation. For example, the obligation for "Alice" to send "Bob" the "dataset_1" exists only when "Bob" requests or when an external event happens such as an emergency. If this environmental pre-requirement is satisfied, the corresponding obligation will be fulfilled, and the auditors can authorise the request.

4) Operational Conditions: Operational conditions include 1) Conditions by Purpose [3], 2) Spatio-Temporal granularity [3], and 3) Time period. The "purposes" is the aim or intention of the operation, it can be "Commercial Use", "Public Safety Use", and "Research Use", etc. "Spatio-temporal granularity" regulates the the frequency of the certain operation, like "Secondly", "Minutely", "Hourly", or "Daily", etc. "Period" determines how long will the operation being executed.

D. Example policy

A simple example of policy is shown in table III, where "Alice" is obliged to send dataset to "Bob" when "Bob" requests. The "Operational Conditions" regulate that the dataset can only be used for research in 2020. When perform this obligations, the authorisations from "Auditor_1" and "Auditor_2" will be needed.

Now we introduced the manifest and the components of policy translation, we can then explain how to apply them in the auditing process. The following sections show how we use these policies in the auditing process to establish secure and trustworthy data sharing.

Components	Value
<Authorizations>	*Auditor$_1$ and Auditor$_2$*
<Obligation>	*Alice is obliged to send dataset to **Bob***
<Environmental Condition>	*With the request from Bob*
<Operational Conditions>	*<**Purpose**> Research <**Period**> In 2020 <**Spatio-temporal granularity**> By default*

TABLE III
AN EXAMPLE OF A POLICY

III. AUDITING PROCESS MODELLING BY JASON

In Sec. II-B, we introduced the role of the “auditor”, and in this section we show how we realise it using the AgentSpeak language Jason. Jason is an AgentSpeak language, which can model the interactions of agents within a specific environment. The BDI model of agency embodied by AgentSpeak allows the agents have the properties of autonomy, proactivity, reactivity, and social ability. More concretely, all the agents have their beliefs, which is information that the agent has about the world; they also have desires which are the options that they might choose; finally, the intentions are concrete actions or plans that the agents decided to work towards. However, there is a gap between desires and intentions, the reasoning process. In this work, we apply the procedural reasoning system (PRS) [19, 4] to make agents select between desires(options), which is shown in Fig. 2. In Jason, the agent has access to sensor data from the environment which acts as input to help them form or update their beliefs and desires. For example, the following excerpt of ’Alice”s belief base tells us that agent ’Alice’ has transferred ’dataset_1’ to ’Bob’:

```
//An example of belief
transfer (bob, dataset_1 ) [source( alice ) ]
```

Setting desires is also convenient in Jason:

```
//An example of desire
!announce(bob)
//Define the plan for this desire
+!announce(Agent) : transfer (Agent, Data)
<-
. print (Agent, ' has already received ', Data).
```

In this example, the desire(plan) is to announce that the dataset has already been received, and then print this announcement out. As can be observed, a pre-condition is also implied here, that the “Data” has already been transferred. Only when the pre-conditions being satisfied, can the desire become an intention and the concrete plan be executed. Further, the pre-condition can not only be a belief, but also can be a combination of a belief with a designed rule like:

```
//An example of rule
already_receive (Data) :- transfer (Agent, Data) & Data ==
      dataset_1
//Define the plan for this desire
+!announce(Agent) : already_receive (Data)
<-
. print (Agent, ' has already received dataset_1 .').
```

Now the rule is: if “Agent” has already transferred “dataset_1”, then “already_received(Data)” will be TRUE. Only then the desire will be activated and executed as an intention and print out the announcement.

The main reason to choose Jason for modelling the auditing process is that its features can realise the functions of auditors well. There are two requirements for auditor modelling, firstly, the auditor agents need to be reactive, which means being responsive to changes in the environment. For example, only when an auditor is required to give authorisation will this auditor respond accordingly. Secondly, auditors should be autonomous and have the ability to reason and judge independently whether a request is in line with the policy to generate the appropriate output. The BDI model and PRS make it convenient for us to meet these requirements. The requests, the action log, and the policies can be used as inputs to build the agents’ beliefs. Then, the auditors will judge if they need to audit the requests, if they need, they will further ensure whether to give authorisations or not. These possible reactions are agents’ desires, after the reasoning step, they can generate the corresponding intentions. This process is the reflection of BDI model.

In the PRS, the reasoning step is based on those mentioned beliefs and designed rules. For example, choosing to give authorisations or not is based on rules and beliefs. The rule is comparing the operations in the requests with the policies in their belief, if there is no conflict, then they will output authorisations, and vice versa. The auditors are thus able to reason and react in a proper way.

Accordingly, the BDI model together with PRS allows agents to react autonomously, reactively, and reasonably according to the concrete environment, making them suitable to realise the audit functions. While Jason, this AgentSpeak language, is based on the BDI architecture, we just need to add beliefs and goals to agents, the interpreter will use this belief base without explicit programming. The PRS of the agent is done according to the plans that the agent has in its plan library. We can thus realise the function of auditors by Jason effectively.

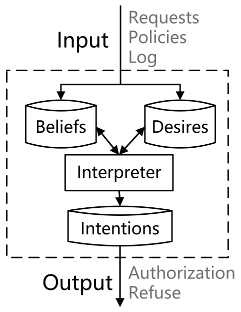


Fig. 2. The procedural reasoning system (PRS). The PRS of agents enables them to generate intentions from desires based on beliefs(rules), and thus react to the environment autonomously.

In section Sec. IV-A, the concrete auditing process applying Jason will be presented.

IV. The data auditing process in ArenA use case

This work maps and enforces the data usage control policies onto multi-domain infrastructures based on a concrete use case which we refer to as the ÁrenA case. In this section, we will first explain the ArenA case in Sec. IV-A, and then realise the data transfer auditing process by Jason, the concrete modelling process is presented in Sec. IV-B.

A. ArenA use case

Johan Cruijff ArenA[1] is the main stadium of the Dutch capital city of Amsterdam [2]. In 2018, on the 26th of May, during the outflow of more than 60.000 visitors from ArenA around 23:30, there was a fatal accident that happened at the pedestrian bridge that leads to parking lot P2, from where someone has fallen off.

Since the accident happened during outflow, conditions were urgent, there are several tasks that need to be done simultaneously, 1) dispatch ambulances to the incident; 2) divert traffic; 3) guiding outflow visitors; 4) clean the scene. To deal with such major and complex incidents, emergency workers from various departments such as *the police, the traffic, and the fire department*, and the ArenA Operational Mobility Center(OMC) are expected rapidly exchange information. More concretely, in this case, the police was responsible for giving directions to emergency services and maintaining the order during the incident; the traffic coordinators from Traffic Management Operations Amsterdam(VMCA[3]) were in charge of diverting traffic away from the incident; the stewards from the ArenA were dispatched to re-route the 7000 pedestrians who needed to cross the bridge to P2 towards their vehicles; the firemen and policemen needed to clean the scene of the accident. Thus, different departments cooperate in dealing with this accident and may have to share data. However, each department needs to remain in control over their own collected data, and must be able to revoke access from collaborators after the situation is under control.

1) Data transfer in the ArenA case: There are three data flows between the departments as table V shows: 1) the ground police informed the OMC about the incident; 2) OMC informed the control room of ArenA at the first moment, so that ArenA control room can dispatch the on-call ambulance to the incident and rescue the wounded within 5 minutes; 3) While re-routing traffic, the parking data from the OMC, who owns real-time data of the parking lot, is transferred to the VMCA so that the VMCA can make decisions on how many people are parked and still need to leave the area.

[1]https://www.johancruijffArenA.nl/home.htm

[2]https://www.johancruijffArenA.nl/Stadion-omgeving/Spreekbeurt.htm

[3]https://nonoa.nl/projecten/verkeersmanagement-centrale-amsterdam

Data	Sender	Recipient	Way
Alarm	Ground police	OMC	Offline
Ambulance	OMC	Control room	Offline
Parking lot data	OMC	VMCA	Online

TABLE V
Data flow in the ArenA case

The first two data flow completed offline, and are not involve private data transferring, this work is thus focusing on the third flow: real-time parking data transferring between the OMC to the VMCA for traffic diversion.

2) Data formats: The raw data of the parking lot is in the form of text, for example, the number of cars near various exits and areas. The data is dynamic, thus needs to be shared timely during the who traffic diversion period.

3) Data transfer policy: To ensure privacy and safety, the data sharing process is supposed to be in line with certain policies. More concretely, it has to be regulated clearly *who* is able to use the private data for what *purpose*, *when* the data can be used by the recipient, and what the permitted *operation*s are that can be applied on the data.. In the ArenA case, the policy is described as:

"*Under normal conditions, the parking data is private to the OMC (data controller). However, when the VMCA will require the parking data to divert traffic, then the OMC has the obligation to satisfy this requirement and share the required dataset.*" By applying the policy concepts in Sec. II, this policy can be transcribed as in table VII.

Components	Value
<Authorisations>	*Auditor*$_1$ *and Auditor*$_2$
<Obligation>	***OMC** is obliged to send dataset to the **VMCA***
<Environmental Condition>	*With the request from VMCA*
<Operational Conditions>	*<**Purpose**> Traffic diversion* *<**Period**> Until the diversion task is lifted* *<**Spatio-temporal granularity**> By default*

TABLE VII
An example of a policy

B. Data transfer auditing by Jason

The auditing process can be divided into two parts: 1) sensing the environment which includes the input of requests and policies; 2) reasoning and reacting which contains the judgement process and the output of intentions.

1) Sensing the environment: The input for auditors includes the requests that are pending for auditing and contain the intended operations on the data objects. For each dataset, its manifest regulates the policies that it should be obeyed. Together with the pending request, the manifest and the policies are the input of auditors in the form of beliefs:

```
//Pending request
request(parking1,omc,vmca, traffic_diversion
        ,2020,07,06,23,45,0) ;
// manifest
data_manifest(parking1,omc,policy_1, ,
        ,2020,07,06,23,45,0) ;
// Policies
policy(policy_1, auditor2, parking1, omc, vmca,
        traffic_diversion ).
policy(policy_1, auditor4, parking1, omc, vmca,
        traffic_diversion ).
```

As shown in the code, in the **request**, it represents "*The OMC will transfer dataset parking1 to the VMCA for traffic diversion, and this request is generated at 23:45:00 on 6th July, 2020*". In the **manifest**, it records the information of the dataset "parking1": it's data controller is "OMC", the policy that should obeyed is "policy_1" and since this dataset is both generated and saved at OMC there is no sender and recipient specified at this moment. Except for the request and manifest, the belief of auditors also include policies that record the obligations and indicate which auditors are required to give the authorisation. In the example, "*policy_1*" regulates that "*auditor_1*" and "*auditor_2*" need to audit that the obligation, that the "*OMC*" should send the "*parking1*" data to "*VMCA*" for "*traffic diversion*" purpose, is properly enforced. Because of these beliefs, only when the auditors' names appears in the policy, they will react and go to the next reasoning and judging stage.

2) Reasoning and Reacting: In this stage, the auditors will check if the "*request*" is in line with the policies, including the data object, the sender, the recipient, purpose, and so on. Then the auditor will judge if the request should be authorised or not.

```
//Reasoning rule − Authorisation
authorisation (Dataset_r,Sender_r,Recipient_r,Policy_m,
      Purpose_r,auditor) :−
    policy(Policy_id, Pointed_auditor, Data_object,Sender,
          Recipient,Purpose)
    & Policy_m == Policy_id & auditor == Pointed_auditor &
          Dataset_r == Data_object & Sender_r == Sender &
          Recipient_r == Recipient & Purpose_r == Purpose.
//Reasoning rule− Refuse
refuse(Dataset_r,Sender_r,Recipient_r,Policy_m,Purpose_r,
      auditor) :−
    policy(Policy_id, Pointed_auditor, Data_object,Sender,
          Recipient,Purpose)
    & Policy_m == Policy_id & auditor == Pointed_auditor
    & (Dataset_r \== Data_object | Sender_r \== Sender |
          Recipient_r \== Recipient | Purpose_r \== Purpose).
```

Finally, the auditors will output if the request is authorised or refused. The following code shows the output process:

```
// Output−Authorisation
+request_auditors(Dataset,Sender,Recipient,Policy,Purpose)[
      source(Agent)]:
    .my_name(Auditor_Name)
    & permission(Dataset,Sender,Recipient,Policy,Purpose,
          Auditor_Name)
        <− .print("This request has been authorised by ",
              Auditor_Name);
            !authorised(Dataset,Sender,Recipient,Policy,
                Purpose,Auditor_Name).
+!authorised(Dataset,Sender,Recipient,Policy,Purpose,
      Auditor_Name)
    <− .broadcast( tell, authorised(Dataset,Sender,Recipient,
          Policy,Purpose,Auditor_Name)).
// Output−Refuse
+request_auditors(Dataset,Sender,Recipient,Policy,Purpose)[
      source(Agent)]:
    .my_name(Auditor_Name)
    & refuse(Dataset,Sender,Recipient,Policy,Purpose,
          Auditor_Name)
        <− .print("This request has been refused by ",
              Auditor_Name);
        !refused(Dataset,Sender,Recipient,Policy,Purpose,
              Auditor_Name).
+!refused(Dataset,Sender,Recipient,Policy,Purpose,
      Auditor_Name)
    <− .broadcast( tell, refused(Dataset,Sender,Recipient,
          Policy,Purpose,Auditor_Name)).
```

Only when the request is authorised by all auditors, the request can be executed. We show a simple authorized example as following:

```
[auditor2] This request has been authorised by auditor2
[auditor4] This request has been authorised by auditor4
[actor] Request: Transfer parking1 from omc to vmca for
        traffic_diversion is authorised by auditor2
[actor] Request: Transfer parking1 from omc to vmca for
        traffic_diversion is authorised by auditor4
[actor] Request: transfer parking1 from omc to vmca for
        traffic_diversion can be executed.
[actor] No requests need to be audited.
```

To conclude, using this process, each request for certain datasets will be checked by the auditors to guarantee that the request is in line with the policies that apply to the dataset. And only when all the auditors authorise the request the action can be executed on the infrastructure. Thus, this process ensures that the policy is enforced, that the operations are trustworthy, and that the security of datasets is guaranteed. The concrete description of the proposed infrastructure is elaborated in Sec. V.

V. Infrastructure

To operationalise applications and their accompanying policies we need an multi-domain infrastructure that brings together all the necessary parts including the compute resources, networking capabilities, policy control, administrative domains, monitoring and the actual applications. In order to keep the infrastructure loosely coupled with different interacting administrative domains, we model the infrastructure as a set of actors whereby actors from different domains interact with each other to run the applications. Actors are able to communicate between each other using an overlay network.

The main purpose of a network overlay is to bring together these actors. In network terminology actors are synonymous to nodes on the network thus a network node can be considered the implementation of an actor. For the rest of this paper actor and node are used interchangeably.

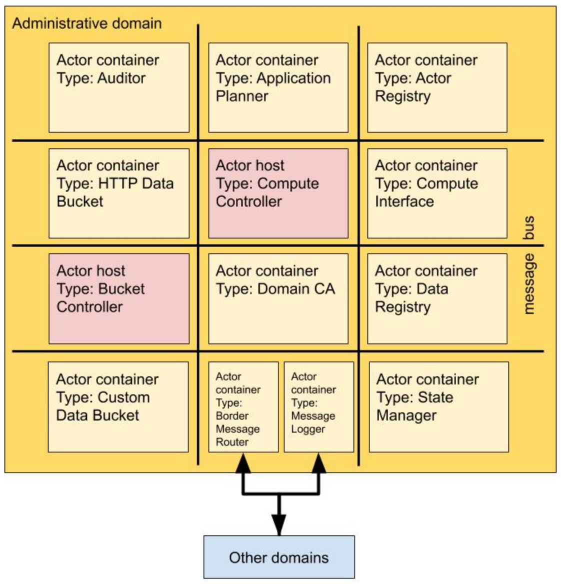

Administrative domain
Actor container Type: Auditor
Actor container Type: Application Planner
Actor container Type: Actor Registry
Actor container Type: HTTP Data Bucket
Actor host Type: Compute Controller
Actor container Type: Compute Interface
Actor host Type: Bucket Controller
Actor container Type: Domain CA
Actor container Type: Data Registry
Actor container Type: Custom Data Bucket
Actor container Type: Border Message Router
Actor container Type: Message Logger
Actor container Type: State Manager
message bus
Other domains

Fig. 3. Actors as container nodes in a single administrative domain. Communication between intra-actors is done on the local message bus while inter domain actor communication is done through the border message router.

Being an overlay, all nodes are able to communicate with each other and communication is governed by policies. For effective communication between nodes an addressing scheme is needed. The addressing has the responsibility of identifying the domain, node and function so that any actor can call functions on other actors from different domains. Addressing on the network is done using public/private keys where the public key is the address of the node and the private key is only known to the node. The advantages of this cryptographic addressing mechanism are twofold: First, nodes are not tied to IP addresses and thus can be dynamically allocated to different physical locations. Second, communications from nodes are signed by their private keys which can be verified by any node in the network since the address (public key) is used to verify the signatures. The latter is important since a core function of the network is creating an audit trail of activity on the network and audits need to verify to whom communications belong. Unlike other addressing schemes such as IP, these cryptographic addresses are non-transferable i.e. the same address can not be reused for different nodes at any point in time which strengthens traceability since any actions signed by a node can always be traced back to that node. Name services are responsible to translate public key address to actual IP endpoints to allow communication to proceed over lower protocols. Another side effect feature of such addressing is that nodes can sign each others keys creating a web of trust between nodes that can easily be verified by following the signatures.

Nodes on the network expose functions which can be called by other nodes. The function routes takes the form:

$$hDPK/hAPK/mN/fN$$

where **hDPK - hashDomainPublicKey** is the public key of the domain root level certificate. The domain root certificate is used to sign actor addresses for a single domain this allows any actor to verify to which domain an actor belongs. Hashing the key allows us to minimise the length of the address which is limited on most message queue systems. The domain public key is also used as the border domain message router which is responsible of routing messages to the different domains based on the domain public key identifier.

hAPK - hashActorPublicKey is the address of any node/actor on the network. This address is generated and signed by the responsible domain. The key is hashed to minimise the length of the routing key.

mN - moduleName is a categorisation defined by the actor. The actor is responsible of structuring its own modules. This information is published to the actor registry when a node is deployed.

fN - functionName is the name given to a callable function on the actor. Together with the module name, this provides a unique way to call functions on actors. This information is also stored in the actor registry so that is can be discoverable.

Figure 3 depicts the high-level multi-domain actor architecture. The basic concept is that actors are represented by containers. Container technologies provide several advantages namely software and network isolation. Thus each actor can be implemented and deployed independently of the other actors. Furthermore containers are conducive to software attestation since the versioning control assures that verified version of the software stacks are deployed on infrastructure. In our system actors have types and types have different functionalities. Figure 3 depicts a basic set of actor types but the architecture is in no way limited to these types. Each domain can define and deploy custom types which come into play when dealing, for example, with different data transfer protocols. The actors fall into four broad categories i.e. actors for authorisation and authentication, actors for control layer, actors for data layer, and actors for compute layer. The current types we envision are:

Auditor actor forms part of the authorisation layer which is one of the main components of the network. Authorisations are done on the bases of policy. Each auditor, internally, executes a belief and desire system as described in Sec. III. The Jason model listens for application requests on the message bus and authorises them by signing the request depending on its belief and desire model. Each domain can have as many auditors as needed and it is up to the agreed application on how many and which auditor signatures are needed for each application to go ahead. The distribution of the auditors allows decentralisation of authority which in turn distributes trust onto the multiple domains.

Application planners are actors that coordinate the execution of the distributed application. We model applications as workflows that are translated to a list of instructions and transactions that coordinate the application. Fig. 5 illustrates a high-level view of an workflow application and its equivalent list of transactions. The figure also illustrates the notion of execution archetype. Application archetypes define the pattern an application should follow. For example, moving compute to data or moving data to compute are two different archetypes of the same application logic. The choice between archetypes is a matter of pre-agreed policy. Planner actors are responsible to get authorisations from the necessary auditors for each instruction/transactions. These are done through requests as described in Sec. IV-B. A planner sends a request for signatures on the message bus to the requested auditors. Upon receiving the signatures the planner can go ahead and contacting the control actors such as data buckets to make the data transfers.

Data registries are responsible for publishing and maintaining the data catalogues. They act mainly as a query endpoint for discovering new datasets and contain infrastructural information e.g. which bucket controller is responsible for which data manifest.

Data bucket controllers are controllers that run on the host machine. This is because they are responsible of booting containers an managing network interfaces for the data buckets. When data needs to be transferred a bucket controller is responsible for creating an data endpoint container known as a data bucket and create the service within the bucket such as an HTTP server to serve the data. The bucket controller is invoked by the planner to start data transfers. The bucket controller is responsible for checking that the planner has acquired the requested auditor signatures before going ahead and booting up the data buckets. Bucket controllers from different domains need to coordinate to open VPN tunnels for data transfers. The tunnel interfaces are attached to the data bucket container at runtime.

Data buckets are the actor containers that are responsible for the transferring of data. The buckets are transient and only created by the bucket controller when a transaction is invoked. This minimises the exposure of the data which minimises the attack vector. The data buckets network interface is controlled by the bucket controller. In essence the network interface only connects to the client data bucket over a dedicated VPN for the transaction. This means communication is only limited to the two data buckets, server and client. Figure 4 illustrates this concept of controller and buckets in a multi-domain scenario.

Custom data buckets are data buckets are are application specific. The creation sequence is similar to the data buckets but differ in the services being hosted in the bucket. For example, a data streaming data bucket hosts different services than an file server.

Compute controllers are responsible for managing the processing part of the workflow. Similar to the bucket controller, the compute controller is responsible of verifying the agreed code is being executed and to configure the resources for compute e.g. using GPU clusters to perform the computation. It is also responsible to create compute interfaces for the duration of the execution e.g. portals and web services.

Compute interfaces are transient containers created by the compute controller to facilitate the application. These are application dependant are custom for each application. For example, an application could expose and API which is accessed by the planner to coordinate the execution of the workflow.

State manager actors are responsible to introduce external events into the network to drive the applications. For example, from the ArenaA use-case described in Sec. IV-A an application is invoked in a state of emergency. The state manager is responsible to broadcast these events on the message bus which will in turn trigger the appropriate planners to start executing.

Actor registries act as address books for the running actors. They serve the discovery of the actors and the functions they are exposing along with other infrastructure details such as public key to IP mappings which is used by controllers to setup connections.

Domain CA is the top level certification authority of the domain. It is responsible to sign keys of its domain actors which in turn is used by other domains to verify that an actor belongs to a particular domain. The different domain CAs form the web of trust and each domain is responsible to include foreign domains as trusted. This is typically done in the negotiation phase of creating the policies. By trusting a domain means that actors from the different domains can communicate.

Domain border message router is the contact point to other domains. The addressing scheme we described earlier includes the hash of the domain public key. This is used by the border router to forward any messages to the correct domain. It also is responsible to verify if the foreign domain is not trusted in which case it ignores the messages. The border router is also responsible to receive messages from foreign domains and put them on the local message bus. Here also, the router verifies if the foreign domain is trusted.

Message loggers maintain a tamper proof database of messages. Since the system has no central authority that controls the logs each domain is responsible to capture enough logging information from other domains. This can be a policy decision such as each domain cross logs with another domain which minimises the possibility of a domain tampering with its own logs. The main reason for maintaining a record of messages is for diagnosing postmortem policy violations. These could occur for several reasons either from wrong implementation of the policy or malicious attempts to thwart the policy.

A. Prototype

To prove our approach we implemented an overlay prototype that we showcased for the first time during the 2019 SuperComputing conference in Denver [6]. Our current prototype shows a secure execution environment that can be used in a multi-domain infrastructure. In particular, we focus on orchestrating virtual infrastructure such that it only allows the

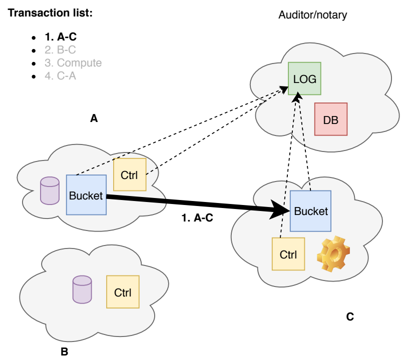
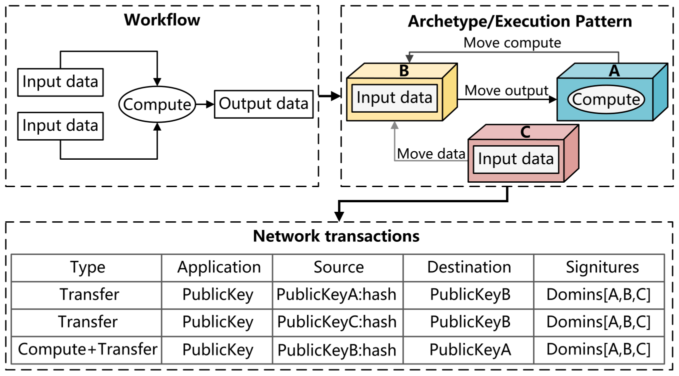


Fig. 4. Three domains A,B,C running controllers. The domains are tasked to execute the transition list (top left). The first transaction (bold arrow) shows a bucket in domain A connected via VPN to a bucket in domain B. At the top right you see the Auditor component, which receives updates from the domains about the transactions occurring



Type	Application	Source	Destination	Signitures
Transfer	PublicKey	PublicKeyA:hash	PublicKeyB	Domins[A,B,C]
Transfer	PublicKey	PublicKeyC:hash	PublicKeyB	Domins[A,B,C]
Compute+Transfer	PublicKey	PublicKeyB:hash	PublicKeyA	Domins[A,B,C]

Fig. 5. Realisation of a workflow application to a set of network transactions.

agreed application transactions to occur and to provide detailed logging that is used during the runtime stage of the auditing process.

All nodes are implemented as web services but instead of reacting to HTTP requests they react to messages on the message bus. Using the concept of topics on the message bus, actors can listen to any message that is directed to them with their public key as part of the topic. The node will then switch internal to call the relevant function. Signing the response with its own private key will ensure that the response came from the actual intended actor and not some other actor nefariously responding to messages.

Buckets are managed using the bucket controller that handles the creation and destruction of buckets. By default, buckets do not have any network connectivity and no interfaces associated to it. When a transaction occurs, bucket controllers use Wireguard [8, 18] to configure an encrypted VPN connection between two buckets and moves the VPN interface into the network namespace [5] that is associated with the bucket. The encryption keys for the VPN are transaction specific thus each transaction is secured separately. When the transfer is completed or when computation is executed the network interfaces are removed from the containers to prevent unauthorised communication.

To maintain a tamper resistant audit log, the prototype borrows some ideas from blockchain. Each logger maintains a linked list of logs using hashes. A set of logs is called a block. Periodically, a new block entry is created by hashing the previous set of logs and including the hash in the next list entry. The list of hashes means that a modification to a log entry will break the hashing. Still nothing stops a malicious attempt to tamper with the log to modify the whole database. For this reason new block hashes are announced to loggers of different domains which they will include them into their own logs (Fig. 6). With this approach logs from different auditors can be cross-referenced for tampering.

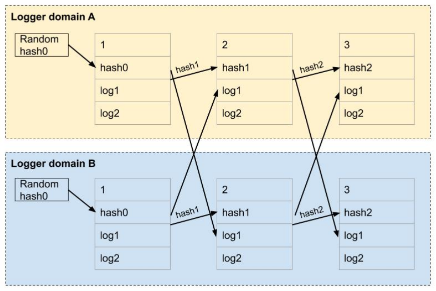


Fig. 6. Auditors maintaining separate audit log in a linked list of hashes to minimise tampering. Block hashes are cross referenced between auditors to prevent audits modifying their own log locally.

Figure 7 depict a running prototype with a hello world workflow. The workflow consists of two data image inputs a compute and an image output. The archetype of this workflow dictates that a third party domain will do the computation thus compute, and two data inputs are transferred to a third domain for computation and the output is copied out as can be depicted from Figure 5.

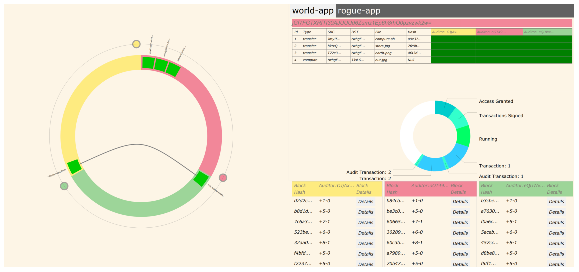


Fig. 7. Left circle: shows the overlay network with data buckets in different colour coded domains. The lines between buckets depict dynamic VPN connections being created per transaction. Top right: the workflow transaction list and the planner address that is coordinating the execution of the transactions. Centre right: the application progress. Bottom right: the 3 auditor logs (1 from each domain) logging and auditing the activity on the network.

B. ArenA applicability

Applying our infrastructure approach to the ArenA use-case (Sec. IV-A) means that both domains, OMC and VMCA maintain their own administrative domain and host several actors that would allow the execution of the applications. The coordination of the data transfer application is encoded and handled by a planner container as list of instructions that

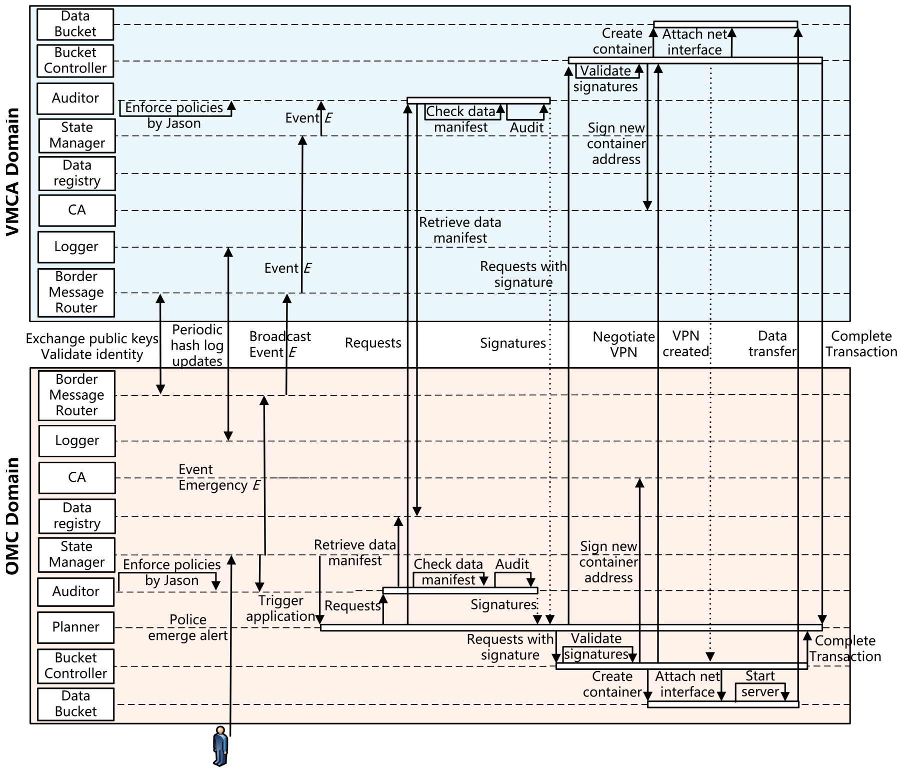


Fig. 8. Sequence of multi-domain actor interactions for realising a data transfer between two domains in the case of an emergency event has been triggered.

are triggered on an emergency event. Each domain hosts an auditor actor which is populated with the belief and desire routines described in Sec. IV-B. Having an auditor actor on both domain allows each request to be signed by both domains before going ahead. The coordination of the data transfers need to a coordinator entity which is encapsulated in a planner actor. The planner actor can technically be hosted in either domain and is matter of agreement between the involved parties who is responsible for which planner. A border message router allows for information flow between the two domains. While bucket controllers allow for configuring the data transfer endpoints.

Figure 8 depicts a high-level view of the minimum interactions between the different domain actors for the dataflows described in Section IV-A1. In the event of an emergency being triggered by the police and event is fired from the OMC's state manager. The event is broadcast to the network and is picked up by the auditors which update their internal system. The state managers of each domain listening on the network also register the event and shift the state of all the internal components to the new state. Planners are triggered on events i.e. the planner to effect the data transfers between OMC and VMCA initiates its execution upon the emergency event. In this case, the planner will first need to get authorisations from the application pre-agreed auditors. A signing request is sent to auditors of the OMC and VMCA domains. At this stage a manifest is built as input for the Jason input (as shown in table I). The auditors can only sign the requests if the output of the Jason model program is positive in which case they will return a signed version of the transfer requests back to the planner. The planner can now instruct the controllers on both domains to start the transfers. Each controller first validates the signatures and then go ahead to coordinated between themselves to setup a transaction dedicated VPN and containerised services to host server/client programs. The transfer can then take place after which the controllers tear down the connection and services. For clarity many other component interactions are not depicted here e.g. logging, address key validation, key generation, and populating registries/catalogues.

VI. Discussion and conclusions

In this paper we described and architecture for collaborative multi-domain applications with an emphasis on policy modelling and enforcing as a core part of the architecture. We showed how we can architect an infrastructure for a real world use-case starting from policy down to the actual network connectivity. Although the application we have used is very simple (simply copying data from source to destination) the fact that has to be done in a multi-domain under different policy quickly explodes the infrastructure needed to support such applications.

The proposed architecture and conceptual model guarantee the enforcement of data policies, and thus realise the secure and trustworthy data sharing process. In the conceptual model, we first introduced the “manifest” to describe the metadata of the dataset, the “auditor” to judge if the request of operation on the infrastructure is compliant with the policies or not, and the components required to construct policies. In addition, we also applied these components to map the policies into conditional statements and executable actions, so that the auditing and authorisation process can be auto executed. In this work, we apply the AgentSpeak language Jason to implement this auditing process which becomes a cornerstone of safeguarding data transfers in our decentralised architecture.

The decentralisation of the architecture allows for no single controlling domain in favour of a more complex web of trusted domains. This means every component such as auditing and logging has to be thought of as a distributed, decentralised component since single control of such components requires trust in a one domain that will no tamper with logs for its own benefit. We architected domains as their own administrative domain and inter-domain communication is done through messaging akin to email services. The actor model approach allows for a scalable decoupled architecture while the cryptographic addressing ensures a secure and identifiable communication.

Although our system aims at enforcing policies in distributed infrastructure, it assumes non-malicious parties. Since domains control their own administrative boundary, they can potentially leak data or tamper with compute. Preventing such malicious attempts is whole other research effort which could include data watermarking and deterministic compute code which can be verified by multiple parties or monitoring container system calls to detect malicious anomalies in the containers code base.

References

[1] Riham AlTawy and Amr M Youssef. Security tradeoffs in cyber physical systems: A case study survey on implantable medical devices. *IEEE Access*, 4:959–979, 2016.

[2] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.

[3] Quyet H. Cao, Madhusudan Giyyarpuram, Reza Farahbakhsh, and Noel Crespi. Policy-based usage control for a trustworthy data sharing platform in smart cities. *Future Generation Computer Systems*, 107:998 – 1010, 2020.

[4] Ge Chu and Alexei Lisitsa. Poster: Agent-based (bdi) modeling for automation of penetration testing. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–2. IEEE, 2018.

[5] J. Claassen, R. Koning, and P. Grosso. Linux containers networking: Performance and scalability of kernel modules. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 713–717, April 2016.

[6] Reginald Cushing, Ralph Koning, Lu Zhang, Paola Grosso, and Cees de Laat. Auditable secure network overlays for multi-domain distributed applications. In *IFIP Networking 2020*.

[7] Simon Dalmolen, HJM Bastiaansen, EJJ Somers, Somayeh Djafari, Maarten Kollenstart, and Matthijs Punter. Maintaining control over sensitive data in the physical internet: Towards an open, service oriented, network-model for infrastructural data sovereignty. In *6th International Physical Internet Conference (IPIC), London 2019*, 2019.

[8] Jason A Donenfeld. Wireguard: Next generation kernel network tunnel. In *NDSS*, 2017.

[9] Johnson Iyilade and Julita Vassileva. P2u: a privacy policy specification language for secondary data sharing and usage. In *2014 IEEE Security and Privacy Workshops*, pages 18–22. IEEE, 2014.

[10] Basel Katt, Xinwen Zhang, Ruth Breu, Michael Hafner, and Jean-Pierre Seifert. A general obligation model and continuity: enhanced policy enforcement engine for usage control. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 123–132, 2008.

[11] Florian Kelbert and Alexander Pretschner. Data usage control enforcement in distributed systems. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 71–82, 2013.

[12] Zeshan Aslam Khan, Edison Pignaton de Freitas, Tony Larsson, and Haider Abbas. A multi-agent model for fire detection in coal mines using wireless sensor networks. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1754–1761. IEEE, 2013.

[13] Andres Munoz-Arcentales, Sonsoles López-Pernas, Alejandro Pozo, Álvaro Alonso, Joaquín Salvachúa, and Gabriel Huecas. An architecture for providing data usage and access control in data sharing ecosystems. *Procedia Computer Science*, 160:590–597, 2019.

[14] Inah Omoronyia. Reasoning with imprecise privacy preferences. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 952–955, 2016.

[15] Jaehong Park and Ravi Sandhu. The uconabc usage control model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):128–174, 2004.

[16] Alexander Pretschner, Manuel Hilty, Florian Schütz, Christian Schaefer, and Thomas Walter. Usage control enforcement: Present and future. *IEEE Security & Privacy*, 6(4):44–53, 2008.

[17] Giovanni Sileno, Alexander Boer, Tom M van Engers, et al. The institutional stance in agent-based simulations. In *ICAART (1)*, pages 255–261, 2013.

[18] Peter Wu. Analysis of the wireguard protocol. 2019.

[19] Gulnara Zhabelova, Valeriy Vyatkin, and Victor N Dubinin. Toward industrially usable agent technology for smart grid automation. *IEEE Transactions on Industrial Electronics*, 62(4):2629–2641, 2014.

[20] Lu Zhang, Reginald Cushing, Leon Gommans, Cees De Laat, and Paola Grosso. Modeling of collaboration archetypes in digital market places. *IEEE Access*, 7:102689–102700, 2019.