Utilizing Remote Evaluation for Providing Data Sovereignty in Data-sharing Ecosystems

Fabian Bruckner
Fraunhofer ISST
fabian.bruckner@isst.fraunhofer.de

Falk Howar
TU Dortmund
falk.howar@tu-dortmund.de

Abstract

The maintenance of digital sovereignty is an important aspect of data-driven business models and data-sharing ecosystems. Considering this, sensitive data is often stored in proprietary systems under the data owner's control and with appropriate security mechanisms. However, nowadays, it is often necessary to share data. As executing unknown and untrusted code on systems containing sensitive data is potentially dangerous, data-processing algorithms cannot be directly sent to the data-storing systems, as one solution. Instead, we have implemented an approach called remote processing that uses the domain-specific language D°, which provides built-in usage control mechanisms for data processing tasks. The approach extends the well-known remote evaluation paradigm that allows controlled, distributed data usage without actual data sharing (transmission via network). Instead of classified data, applications and their execution results are transmitted. This way, sensitive data is never directly exposed to third parties. Furthermore, the application-integrated usage control mechanisms prevent malicious data usage.

1. Introduction

One central part of Industry 4.0 is the integration of information technology into machines, used for industrial production, which results in enormous amounts of collected data amongst other things. The ongoing transition to industry 4.0 results in new requirements for data protection and data usage control. As the new data itself is increasingly becoming a valuable good, it is necessary to protect it. Innovative business models that mainly rely on the handling and processing of data are emerging in the context of industry 4.0 [1]. Some known examples of such business models are predictive maintenance and smart supply chain management. For many companies the owned data is one of the most valuable, if not the most valuable asset and critical to their business models. Therefore, reliable protection is required for these assets, especially if the data needs to be shared with third parties. In particular the ability to make independent decisions about the use of one's own data is necessary. This ability is called data or digital sovereignty [2] and includes, for example, the ability to restrict the purpose for which data is used, to limit the period of use.

It becomes apparent that reliable mechanisms for protecting data and regulating its use (by third parties) are necessary.A technical solution to these issues is provided by usage control. Usage control is "a fundamental enhancement of the access matrix" [3] and in this context, many different approaches have been developed. Existing solutions allow to add usage control mechanisms to existing software or to integrate them into newly developed applications. LUCON, for example, is a label-based approach that allows to track and restrict the usage of data within distributed systems [4]. Correct usage control is often quite challenging [5]. This issue can be addressed by the policy-agnostic programming paradigm. The paradigm is based on the separation of application logic and policy enforcement during the development and later combination into one application. The correct combination of these two elements can be automated.

However, these solutions have a potential weakness that might cause problems in some usage scenarios. They are not well-suited for use cases where data should be shared with a third party.Data is transferred to a third party and processed by its software and systems that feature usage control mechanisms. In these scenarios, the data is transferred to systems that are not controlled by the data owner.Once the data leaves the data owner's system, it is no longer under the data owner's control. Even with contractual agreements and technical solutions for usage control, this can result in unallowed data access and usage as soon as a malicious administrator of the receiving system bypasses the usage control mechanisms and ignores contracts to gain full access to the data.Furthermore, the transmission of data

URI: https://hdl.handle.net/10125/71463
978-0-9981331-4-0

via network is a possible point of attack.

In order to address these issues, we developed a technology called remote processing which extends the remote evaluation paradigm from the area of mobile code. Remote processing reverses the direction of transmission. The sensitive data stays in the owner's system and the data processing application is transferred to the data owner's system as source code. The data owner has the possibility to audit the received code before it gets compiled and executed in a sandbox environment. The data processing entity will only receive the results which are produced by the sent application. That way the sensitive data never leaves the systems which are controlled by the data owner.

In this paper we introduce the remote processing system. The system allows the reception of source code, its compilation, execution, and removal being controlled by the sender after the recipient has approved the application. The system allows the cooperative data usage with third parties while protecting the sovereignty of the data owner. The received applications are executed in application containers in order to provide isolation from the executing system as well as the application provider. We demonstrate the implementation of the system for the programming language D° (pronounced di'grē). D° is a policy-agnostic domain specific language with integrated usage control mechanisms [6]. We show how the data owner can add additional usage control policies to received applications if programming languages with embedded usage control are used for remote processing. The used architecture allows easy extensions in order to allow the usage of various programming languages with integrated usage control functionalities. The remote processing system is an extension of the remote evaluation paradigm and integrates usage control mechanisms as a fundamental system component that allows to address the aforementioned requirements of industry 4.0 and data-driven business models.

1.1. Related Work

Remote execution of applications is known from distributed systems where powerful centralized or idle hardware is used by other machines with potentially very weak hardware [7]. The approach has been adapted for different device types (e.g. mobile) [8] and improved for classic thin-client architectures [9]. Another approach provides a JavaScript framework that allows the flexible distribution of web applications to different devices [10]. However, these approaches do not target the secure processing of data at its source with regard to usage control. Instead, they provide computing power for devices that exceed the device's own hardware capabilities or distribute the computation across multiple devices.

The proposed system is more comparable to mobile code systems [11]. Remote processing is an extension of the remote evaluation paradigm, which is one type of mobile code [12]. The basic idea of sending applications via network, executing them on a remote machine, and returning the execution result is extended by modern usage control mechanisms to meet the challenges of data usage control that arise nowadays [13].

There are several existing (domain specific) programming languages that feature different types of usage control mechanisms. To extend the Java language with the possibility of statically checked information flows, JFlow was developed [14]. There, an additional byte code checker allows to perform information flow checks on source code as well as byte code [15]. By extending the Java programming language, users of JFlow are themselves responsible for ensuring that the provided usage control mechanisms are used correctly. The used programming language D°, however, is cross-compiled into a general-purpose programming language that is used as host language. Therefore, users of D° cannot omit the usage of the language's policy system, neither accidentally nor intentionally. In addition, D° implements the policy-agnostic programming paradigm in order to simplify the usage of the policy system significantly.

The policy-agnostic programming paradigm was introduced by Yang et al. and implemented in the programming language Jeeves [16, 17]. It is based on the separation of application logic and policy enforcement. These two components must be combined at a later stage of development (e.g. during compilation) to create an application that contains the application logic and complies with the defined policies. In contrast to D°, Jeeves is characterized by the modification of data in situations where execution has to be stopped due to unfulfilled policies. If a modification of the data would allow the execution, Jeeves is capable of doing so. D° does not provide this feature since it can lead to unwanted behavior in data processing applications.

LIFTY is another domain specific programming language that implements the policy-agnostic programming paradigm which can be used for the development of data-centric applications [18]. In order to combine application logic and policy enforcement into a single application, LIFTY uses program synthesis to perform information flow and (unintentional) information leak checks during compile time. This is contrary to D°, which uses techniques from the field of model-driven software engineering (e.g. code

generation) to embed a policy system into the compiled application that is operating during the runtime of the application.

1.2. Outline

The remainder of this paper is structured as follows. Section 2 presents how the remote processing technology is designed and how the different aspects address the mentioned goals. The procedure to make data usable for applications within the remote processing is described in Section 3. A simple example is used to demonstrate the proposed technology in Section 4. A discussion and conclusion of the remote processing technology is given in Section 5.

2. Design

This chapter presents how the remote processing system is structured. The different states of the process which define the remote processing technology are described. In addition some aspects of our realization of remote processing are shown.

The different aspects of the remote processing were created by using the prototyping method according to Alavi [19].Between the different prototyping iterations, new and adapted requirements for the remote processing emerged in discussions with experts.

2.1. Process

The process of remote processing can be described roughly in four different steps.

1. Transmitting the source code including all resources that are required for compilation and executing to the remote processing system.

2. Performing an automatic audit of the received application and extending it by usage policies defined by the system's operator and providing data which can be used by the application.

3. Compiling the application and deploying it by using an application container.

4. Executing the sandboxed application and returning the execution's result.

Figure 1 shows a schematic diagram containing the different components of remote processing, grouped by functionality.In addition, the upper part of the figure shows the most important states of remote processing as a process. A complete overview of the states of remote processing is given later in this section. The components

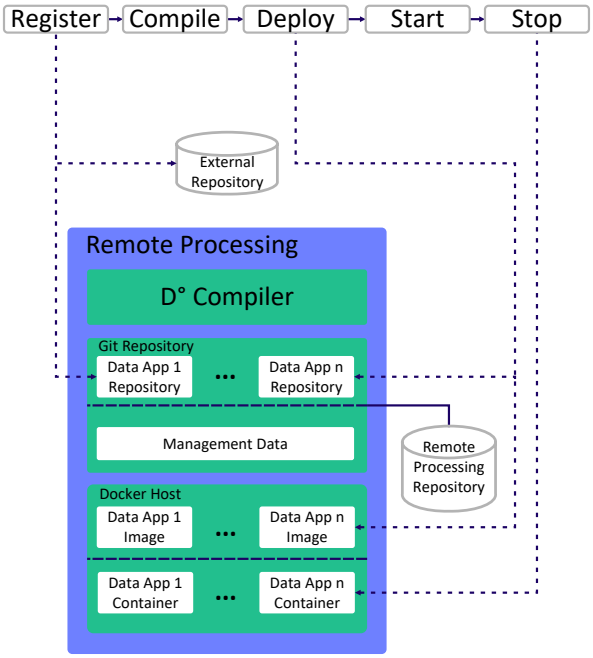

Figure 1. Block Diagram Showing the Components of the Remote Processing System, and Components Which Are Used in Register, Deploy, and Stop Requests.

used in the states register, deploy, and stop are indicated by arrows as a sample.

The main contribution of remote processing is step two of the described process. The steps 1, 3, and 4 basically describe a classic remote evaluation process [12] extended by security mechanisms that are known from the area of mobile code [20]. The added usage control mechanisms are a cross-cutting concern which can be identified in all stages.

However, if we take a more in-depth inspection of the remote processing process, there are various differences to the classic remote evaluation process that is known from the area of mobile code.

The first difference becomes apparent right at the beginning of the process. Instead of source code and required resources, the application provider registers a source code repository that contains all required elements. The application provider can trigger the remote processing system to retrieve an update from the repository.This offers an easy way to provide updates for applications, used in remote processing.In addition, using source code repositories instead of source code allows to track which versions of an application have been used for remote processing.

Next, updated applications must be audited before they can be prepared for actual execution. Since

D°, the programming language used for the remote processing, does not provide any automated code auditing capabilities, the possibilities for automating audits are limited in the current version of remote processing. Future versions of D° will feature automatic code audit functionalities which will allow a more comprehensive automatic audit on the application. Alternatively, existing methods for e.g. static code analysis [21] can be added to the system to add additional capabilities during the audit. Since D° implements the policy-agnostic programming paradigm, it is easy to add additional policies to applications. The separation of application logic and policy enforcement during the development and later combination during compilation, which are the key points of the paradigm, allow to add additional policies without requiring modifications to applications. Therefore, policies that are predefined by the system's operator can be automatically added to the application during compilation.If further audits of the received application are required (e.g. regarding the code quality), they have to be performed manually. The remote processing operator and the application provider must have agreed on which data can be used by the application. How this data is provided to and used by the application is described in Section 3.

Once the audit has been performed, the application can be prepared for its actual execution. This includes the compilation and actual deployment of the application. In case either of these two steps fails, the application provider can retrieve an error message and then decide to either remove the application from the system or to add a solution for the problem to the application's repository and update the application. It has to be noted that each time the application is updated, a new audit is performed because the previously added usage policies are no longer available. In order to ensure isolation of the application and the executing system, the deployment includes building an application container that is used as sandbox for the application.

After the application provider has triggered the execution of a deployed application, the corresponding container is started. Running the application inside an application container ensures that neither the executing system nor the operator's data are directly exposed to the application. Following the starting phase, the embedded application either starts the data processing or provides an API that is used to start the data processing, depending on its type. Self-running applications that operate on the command line as well as applications that provide an HTTP API are supported by the system. If the application provides an HTTP API, the application provider has to call it to start the data processing. The provided HTTP API is only used to start the execution of the application, it cannot be used to alter its behavior (e.g. by parameters). To ensure isolation between the application provider and the application, this API is not exposed.Instead, the remote processing delegates the application provider's requests to the application. Since the application is not exposed to the provider, there is no possibility for hidden malicious functionality which can be used by the application provider. The remote processing system has full control over the application while the provider triggers state changes. Regardless of the application type, the remote processing system provides an endpoint that can be used by the application provider to retrieve the execution results from the applications. The application running inside the container sends a signal to the remote processing as soon as its execution is finished.

If an application will not be used again in the future, the application provider is responsible for triggering the removal of the application.

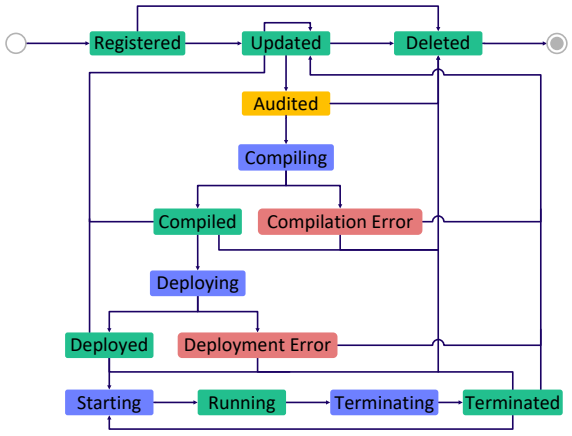

Figure 2. All States and Their Transitions of the Remote Processing Process.

Figure 2 shows all states and their transitions that are used in the remote processing. Green nodes represent states which are entered as a result of an action performed by the application provider or a positive result of a system action triggered by the application provider. These states are only entered and exited if the provider triggers the corresponding action. The two possible exceptions to this are the 'Deleted' state, since an application automatically leaves the remote processing after it was deleted, and the 'Running' state, since the executed application may terminate itself. Blue nodes are intermediate states that are also entered after the application provider has triggered the corresponding action. They are automatically exited as soon as the system has performed a specific action, e.g. compiling

the application or shutting down a running application. Red nodes indicate error states and can only be entered after a system action (blue nodes) has failed. The 'Audited' node is the only yellow block, as it allows optional manual intervention. The state is entered when the application provider requests an audit for an application. Depending on the audit result, the application provider has several options for which state transitions to trigger. If the audit fails, the application can be updated or removed. If the audit is successful, it is possible to compile the application.

2.2. Realization

The remote processing system provides an HTTP API that is used for all interactions with application providers. A unique identifier is assigned to each registered application and necessary for triggering state changes or requesting execution results. Our implementation is using Spring Boot in order to provide the necessary HTTP API.

For each application registered in the remote processing system, the system stores necessary management data such as the location of the repository that provides the actual application code.The management data also contains a list of used commits to allow the tracing of used application versions. In order to persist these management data, the remote processing system can optionally use its own git-repository for storing the data. That way, the different state changes for registered applications can easily be traced and the entire system can be run on different machines without complicated data migration operations. As a result, every application has to be updated when the remote processing system is migrated to another machine.

The D°-compiler is an inherent part of the remote processing system. The system is designed in a way that additional compilers can be added. However, in terms of usage control, this is only useful if the programming languages have embedded usage control mechanisms. Otherwise, the remote processing does not provide any major improvements compared to a classic remote evaluation system. With some modifications to the process, it would be possible to use programming languages without integrated usage control and to deploy them in combination with additional usage control software.However, this is not the preferred solution as it increases the complexity and produces overhead when running an additional application.

To provide the aforementioned isolation between the application and the executing system, our implementation uses Docker to wrap the applications into application containers. It is not necessary for the remote processing system and the Docker client to be operated on the same system. Instead, they can run on different systems and be connected via network. That way, the remote processing provides better scalability. In addition, it prevents the system that executes the applications and has access to the sensitive data, which the remote processing wants to protect, from being exposed to the internet.

3. Data Access

While Section 2 demonstrates the remote processing process one very important aspect has not been described yet: the provision and usage of data which can be used by the applications.

It is hardly intuitive to transfer an application to a remote system and execute it with full access to all data in order to protect the data stored in the system. Therefore, it is necessary to restrict the data that can be accessed by applications. Only data that is required for the application's functionality and agreed upon with the remote processing operator is allowed to be accessed by the applications.

The most appropriate solution depends heavily on the type of data and the way it is stored.For example, files can be easily integrated into the Docker container that is created during the deployment of applications. If the deploying machine is different from the Docker host, volumes can be created and the files, that should be accessible for the application, can be copied into the volume before the container is started.

We have defined patterns for integrating file-based data as well as data stored in SQL-databases. These defined patterns allow an easy way to provide copies of actual data to applications used within remote processing. That way, it is not required to define how data is transferred between the data provider's system and the application's container (and vice versa) on a per-application-level. At the same time, the integrity of the original data is not compromised because the application only gets access to data copies. In addition, the task of providing data to applications is easy for the remote processing operator, since it only has to define which files and/or tables are to be copied into the container. This can be done at the same time as defining the policies to be added to the application. The application provider has to implement the application according to these patterns in order to ensure correct functionality. These patterns define used file names and locations as well as the naming of tables in databases used for in- and output. For example, if data is provided by using a SQL database a SQL dump is used. The location and name of the dump-file as well as used table

names is standardized for remote processing.

Similar patterns can be defined for NoSQL-databases or other data sources, so the approach does not create restrictions to the types of applications that can be used with remote processing. As an additional benefit, the process of providing data to applications does not depend on the internal structure of the data.

If we stick to SQL-data, a similar result could be reached by using individual views with read-only access for the different applications. The benefits of the data provisioning method used by the remote processing are the following:

- Identical provisioning for all kinds of data sources
- Management of data for remote processing takes place completely in the remote processing system
- Various possibilities to filter the data by including other data sources

The application has full control and access to the provided data copy. Therefore it is mandatory to provide only data to the application which may be used by it. Since the application has full control of the provided data, it is necessary to prevent that the application exfiltrates the data by sending it to the application provider. There are multiple aspects in the remote processing which aim at this issue. The usage of containers for the provided applications does not contribute to the data protection by itself. But is ensures that the applications access to data is limited to the provided data copies. An additional aspect is that the application provider does not have direct access to the application, the used container, or the executing machine. The main contribution to preventing data exfiltration is made by using D° as programming language. The policy system, used by D°, allows to define policies for data types as well as activities, which are the atomic functional element in D°. By adding policies which prohibit the exfiltration of the data (e.g. by sending them via HTTP), the compiler creates an application with integrated usage control code which complies to these policies.

Next, it is described how the application provider knows how the data is structured. The internal structure of the data is not defined by the remote processing. These information is required to allow the application provider to develop applications which can use the data. For this purpose, the remote processing operator provides the type definitions from the D° type system. These definitions describe the data which is available to the application in a way that the application provider can use them.

4. Example

For a better understanding of the remote processing system, this section will introduce a usage example.

The medical staff of a clinic is working on a new study. This process involves heavy usage of sensitive patient data.Since this data is very critical, there are very strict regulations regarding the usage of such data and its protection. Due to the current situation with COVID-19, the medical staff working exclusively on the study is working in home office. Because of privacy regulations and data protection requirements, these staff members can no longer access the patient data that is needed to create the studies. In fact, they are not able to work on the study because it is not allowed to copy the data to the staff's computers, which are not appropriately secured.

Since the study does not require the data themselves, but the results of the analyses performed with these data, remote processing can be used to allow the medical staff to continue working on the study. By using remote processing, the medical staff can write data apps with D°, which will process the patient data and produce the results required for the study. The D°-code of these data apps is stored in a repository that is registered at the clinic's remote processing system instance. The clinic runs an instance of the remote processing on its server that is accessible by the medical staff. Access to the remote processing's API is regulated by the clinic's identity provider. The Docker host responsible for the application execution is not operated on this server. Instead, a server of the clinic's intranet is used. The network configuration ensures that this server is capable of accessing the sensitive patient data and is itself only reachable by the clinic's remote processing server. That way, it is possible for medical staff to perform evaluations on the patient data, while ensuring that the data is always resided in the secure systems operated by the clinic or specialized service providers.

The clinic created a definition of the patient data which is made available for applications in the remote processing. A shortened version of this definition can be found in Figure 3. Line 1 defines the name of the data type. Line 2 contains validation rules that are evaluated each time the value of an instance of the type is changed. This ensures that each instance of each data type only contains valid values at any time. The value of the validation is Java code that must return a boolean value indicating whether the new value is valid. The type system allows defined access to elements of the types for validation. Lines 3 and 4 are used to set the data type `MedicalData` as super type for this type. This is important for both type casting and validation. The validation of all super types is performed each time

the validation of the type is executed. Line 5 sets the cardinality of the data type to 0, which represents a list of arbitrary length. Lines 6 to 9 define of which attributes the data type is composed. Lines 10 to 13 contain sample values that must pass the validation (line 11) and must fail the validation (line 13). These values allow an automated check of the given validation code. Line 14 contains a documentation for the data type used for automatic generation of web-based documentation. This allows to auto-generate a documentation for the entire type system and simplifies the usage of defined data types.

```
1  PatientData:
2    validation: "..."
3    supertypes:
4      - "MedicalData"
5    cardinality: 0
6    attributes:
7      - patient: "clinic.Patient"
8      - diagnose: "clinic.Diagnose"
9      - invoice:
           "administration.Invoice"
10     - ...
11   positiveTests:
12     - "..."
13   negativeTests:
14     - "..."
15   wikidoc: "Patient data which can be
         used in studies."
```

Figure 3. The Shortened Definition of the Patient Data Datatype Used in the Example.

The medical staff working on the study may not have the knowledge required to program applications that perform the data processing. Since D° is highly extensible, allowing data types, policies, and activities to be provided as language extensions [6], this problem can be solved. In the context of D°, an activity is an atomic functional block. Each activity contains an arbitrary amount of code in the used host language.Our implementation of D° uses Java as host language. The same applies to policies. Each policy provides its own enforcement code and the code generator, which is part of the D°-compiler, ensures that the code is executed in the right situations.

In order to resolve the aforementioned issue, the clinic has to provide the medical staff with a language extension for D°, which contains all required data types and activities. The contained elements can be used by the medical staff to develop data apps that perform the required data processing. The language extension mentioned does not need to include any special policies for the following reason.

The usage control policies that have to be enforced in order to protect the patient data are added to the applications during the audit phase.These policies are packed as a separate language extension that is available within the remote processing system. That way, the overhead for the medical staff in using D° is minimized. The clinic simply needs to make sure that a set of policies is defined to ensure that patient data is not used maliciously. This set of policies is added to each individual application provided by the medical staff.

The policies which are added to the applications contain policies which ensure that the patient data is not transmitted to other services, used as result without aggregation or anonymization, and preventing access to administration's data like invoices. As an example, the policy which prevents the returning of data without aggregation or anonymization is shown. This policy is important because otherwise the staff could use the patient data directly as the application's return value and bypass the regulations that way.

First, the D°-policy is inspected. Each policy has three different enforcement points: Precondition, postcondition, and the security manager intervention [6]. Pre- and postcondition are executed before and after an activity is executed. The security manager intervention is a construct that allows APIs to be protected and policies to be checked before calls are executed to the protected APIs.

To allow different combinations of language elements (e.g. policies & activities) and static values, D° distinguishes between instances that can be used in D°-applications and definitions that provide the constructs (and implementations) of language elements. Figure 4 contains the policy definition used in the example. Lines 1 to 4 are used to define the name and unique identifier of the policy definition. Since the instance of the shown policy definition does not contain anything special, it is omitted here.

```
1  RequireAggregation:
2    degree.Constraint@RequireAggregation:
3    name:
4      Identifier: "RequireAggregation"
```

Figure 4. The Policy-Definition Which Is Instantiiated in the Example.

The code generator, that is part of D°, ensures that enforcement takes place in required situations [6]. For the example policy, the precondition and postcondition do not contain any enforcement logic. The security manager intervention is used to monitor interactions with the SQL database which is used for reading the patient data as well as storing results. Each time some data should be written to the database, the policy checks if the data is tagged as aggregated or anonymized.

The D° type system allows the usage of tags for data. These tags can be added, changed, and removed during runtime.

Next, we take a look at an application which a staff member develops to analyze the patient data. The person working on the study needs the case numbers within a certain period of time broken down by gender. To get these case numbers, the following steps have to be performed:

1. Load the patient data
2. Filter data that is not within the given time interval
3. Sort the filtered data by gender
4. Count the sorted data

To retrieve these values by using remote processing, the person develops a data app using D°.Since this is a very common query for the given study, the member of the medical staff decides that the data app should provide an HTTP API that can be used by other members of the staff.This can be easily set using the data app configuration, which is part of the D°-code.

Figure 5 shows the data app that implements the aforementioned steps in a D°-application. Lines 1 to 7 contain the configuration of the data app. The exact values for the entries `port` (line 6) and `url` (line 7) are not relevant to the use case, since the remote processing prevents direct access to the running application and delegates all requests to the application's API. Nevertheless, the values are important because they ensure that the application will provide an HTTP API. The tags applied to the application (line 5) can be used within usage control policies. Lines 2 to 4 provide the name, namespace, and version of the application and are used to identify the application.

The actual application logic can be found in lines 9 to 28. Line 10 contains the signature of the data app. The given data app requires a time interval defined by its start and end date as input. Requests that do not contain these two fields are rejected. Lines 11 and 12 as well as lines 24 and 25 generate log messages that can be used to trace the usage of the running application. Lines 14 to 22 contain the implementation of the four required steps mentioned before. All activities which are used in the application are part of the language extension, which is provided by the clinic. Line 14 calls an activity that loads the patient data into a variable which can be used by other activities. As mentioned before the data is provided to the application as a SQL-dump. Because of the defined data provisioning process the location of the dump as well as the names of the tables is known and therefore not required as input for the activity. The

```
1  configuration
2    - namespace : "example.clinic"
3    - name : "genderizedCaseNumber"
4    - version : "1.0.0"
5    - tags: "MEDICAL, STUDY, GENDERIZE"
6    - port : "5000"
7    - url: "query"
8
9  code
10   [startDate = $DateTime,
        endDate = $DateTime] -> begin
11     logMessage = $Text(@write
           ["Starting processing of
           patient data."]);
12     UnconstrainedPrintToConsole
           Activity[logMessage];
13
14     [patientData] = loadPatientData[];
15     [filteredData] = filterByTime
           Interval[patientData,
           startDate, endDate];
16     maleCases = $PatientData();
17     femaleCases = $PatientData();
18     [maleCases, femaleCases] =
           genderizeData[femaleCases];
19     maleCount = $UnsignedInteger();
20     femaleCount = $UnsignedInteger();
21     [maleCount] = count[maleCases];
22     [femaleCount] = count
           [femaleCases];
23
24     logMessage = $Text(@write
           ["Finished processing of
           patient data."]);
25     UnconstrainedPrintToConsole
           Activity[logMessage];
26
27     storeAndDumpResult
           [maleCount, femaleCount];
28   end
```

Figure 5. Service for Processing Patient Data Developed With D°.

activity call in line 15 applies a filter that uses the specified time interval as criteria. Lines 16 to 18 break down the filtered data into gender. Finally, lines 19 to 22 count the filtered and sorted data. Like all activities that are used in the application, the `call`-Activity is part of the D° language extension which is provided by the clinic. This activity also adds the required `AGGREGATED`-tag to the data, which allows the usage as output of the application. Each activity has the possibility to modify the tags which are attached to data. This is a functionality which is provided by the type system of D°. The tags can be used in various places, for example within policies. The result is written to the database in line 27. In addition this activity creates a SQL-dump of the result tables. This dump is used by the remote processing system to provide the execution result to the application provider.

D°-applications always contain application logic and

configuration in the same file. This way, the data apps are self-contained, at least when they do not require special language extensions that need to be provided because they are not already known to the compiler.

Next, the described D°-application is registered at the remote processing system and an update is performed. That way, the source code is available for the audit. Two things happen during the audit:

1. The patient data that is required for the study is dumped into a format that can be loaded by the in-memory database that is used inside the container holding the D°-application. Since the staff needs to perform analysis on arbitrary parts of the patient data, the entire data set is dumped into the containers and can be used by the contained applications. As the medical staff is allowed access to the entire data set, additional filtering of the data that can be used within the D°-application is not necessary.

2. The policies that ensure the protection of sensitive patient data are incorporated into the application. The set of policies defined by the clinic to ensure the protection of patient data is added to the activities used in the given application. The policies could have been added to the language extension for all activities, but since the staff can add their own language extensions to the D°-application, it has to be ensured that the correct policies are combined with all elements. The compiler checks for duplicates later to avoid policy redundancies on elements.

Once the audit is finished, the application is ready for compilation and deployment. When the medical staff triggers the compilation, the application, the clinic's language extension, and the policies, that were added during the audit, are used as input to the compiler. The compilation is a two-phase-process: First, the D°-code is cross-compiled into the host language. The host language code is then compiled into an executable format. The result of this compilation is an executable application that contains all defined policies and their enforcement logic as an inherent part.

After the compilation has been successfully finished, the staff can start to deploy the application. A prepared Dockerfile is used to build the application container which contains all required artifacts. The script ensures that the application is started. When the application terminates, a message is sent to the remote processing system.Since the application in this example is designed with an HTTP API, this signal should never be sent. The execution of the application can only be terminated by the staff.

When triggering the execution of the application logic, the application provider gets a unique identifier. This identifier is used to identify a specific execution of the application logic. The remote processing stores and reuses the identifier. At regular intervals, the remote processing system queries the running application for all known identifiers to check whether the execution has finished and the results can be retrieved by the user.

For applications that do not provide an API (e.g. HTTP) used to trigger the execution, the remote processing system regularly polls the state of the running application and once the execution results are available, they are available for retrieval by the application provider. A unique identifier is also used for these applications to retrieve the execution results.

5. Conclusion

We have shown that the well-known remote evaluation paradigm can be extended by modern usage control technologies in order to help data owners maintain their digital sovereignty. We demonstrated that, by using a programming language with integrated usage control mechanisms combined with code audits, data owners have the possibility to ensure that their data is only used in accordance with agreed rules. Furthermore, we showed that the overhead of using the language's usage control mechanisms can be reduced by using a policy-agnostic programming language.

Since the used programming language D° does not feature automatic code audits, the automatic remote processing auditing is currently limited to adding policies and data to applications. There are several approaches that partially automate code audits, thereby reducing the amount of code that has to be manually audited in situations where a more detailed audit is required [22]. By integrating such mechanisms into the D°-compiler, the functionality of the automatic code audit within remote processing could be improved. If the compiler has access to a machine-readable version of the agreed data usage restrictions and a set of predefined policies that address these restrictions, the compiler could identify relevant parts of the application and add the required policies. That way, the task of applying policies to the application can be limited to a verification of the policies applied by the compiler. The same principle could be applied to the provision of data to applications. By using standardized machine-readable definitions for data to be accessed, the remote processing operator only needs to check the correctness of the data provided by the compiler to the application. The fact that each update of an application potentially results in a new audit can be problematic for the remote processing

provider if there are many updates for applications. In order to minimize the amount of work required, the compiler could be extended to consider the repository history and the results of previously performed audits. That way, unmodified and already audited parts of the application can be identified and excluded from further audits.

The remote processing approach aims at protecting the data and establishing the digital sovereignty of the data owner. Depending on the application which is used to process the data, the algorithms inside the application can also require protection from access by third parties. Such algorithms may contain business secrets or reveal internal information like the way of working. Therefore, the remote processing shifts the problem of using assets that require usage and access control from the data owner to the application provider. This is not within the scope of the remote processing, but has to be considered before usage. Nevertheless, remote processing allows for cooperative and sovereign data usage in many applications.

Acknowledgments

This work was developed in Fraunhofer-Cluster of Excellence "Cognitive Internet Technologies".

This research was supported by the Excellence Center for Logistics and IT funded by the Fraunhofer-Gesellschaft and the Ministry of Culture and Science of the German State of North Rhine-Westphalia.

References

[1] A. Zolnowski, T. Christiansen, and J. Gudat, "Business model transformation patterns of data-driven innovations," 2016.

[2] M. Jarke, B. Otto, and S. Ram, "Data Sovereignty and Data Space Ecosystems," *Business & Information Systems Engineering*, vol. 61, no. 5, pp. 549–550, 2019.

[3] R. S. Sandhu and J. Park, "Usage control: A vision for next generation access control," in *International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, pp. 17–31, 2003.

[4] J. Schuette and G. S. Brost, "LUCON: data flow control for message-based IoT systems," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 289–299, 2018.

[5] P. V. Rajkumar, S. K. Ghosh, and P. Dasgupta, "Application specific usage control implementation verification," *International Journal of Network Security and Its Applications*, vol. 1, no. 3, pp. 116–128, 2009.

[6] F. Bruckner, J. Pampus, and F. Howar, "A Framework for Creating Policy-agnostic Programming Languages," 2020.

[7] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the v-system," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 5, pp. 2–12, 1985.

[8] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *Proceedings of the 1st international conference on Mobile systems, applications and services*, pp. 273–286, 2003.

[9] R. A. Baratto, L. N. Kim, and J. Nieh, "Thinc: a virtual display architecture for thin-client computing," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 277–290, 2005.

[10] T.-L. Tseng, S.-H. Hung, and C.-H. Tu, "Migratom. js: a javascript migration framework for distributed web computing and mobile devices," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 798–801, 2015.

[11] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on software engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[12] J. W. Stamos and D. K. Gifford, "Remote evaluation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 4, pp. 537–564, 1990.

[13] A. Lazouski, F. Martinelli, and P. Mori, "Usage control in computer security: A survey," *Computer Science Review*, vol. 4, no. 2, pp. 81–99, 2010.

[14] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 228–241, 1999.

[15] G. Barthe, D. A. Naumann, and T. Rezk, "Deriving an information flow checker and certifying compiler for Java," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 229–242, 2006.

[16] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 85–96, 2012.

[17] J. Yang, *Preventing information leaks with policy-agnostic programming*. Dissertation, Massachusetts Institute of Technology, Massachusett, 2015.

[18] N. Polikarpova, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, "Enforcing information flow policies with type-targeted program synthesis," in *Proceedings of the ACM on Programming Languages*, vol. 1, 2018.

[19] M. Alavi, "An assessment of the prototyping approach to information systems development," *Communications of the ACM*, vol. 27, no. 6, pp. 556–563, 1984.

[20] A. D. Rubin and D. E. Geer, "Mobile code security," *IEEE Internet Computing*, vol. 2, no. 6, pp. 30–34, 1998.

[21] M. Kulenovic and D. Donko, "A survey of static code analysis methods for security vulnerabilities detection," in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1381–1386, IEEE, 2014.

[22] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*, pp. 797–812, IEEE, 2015.