UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

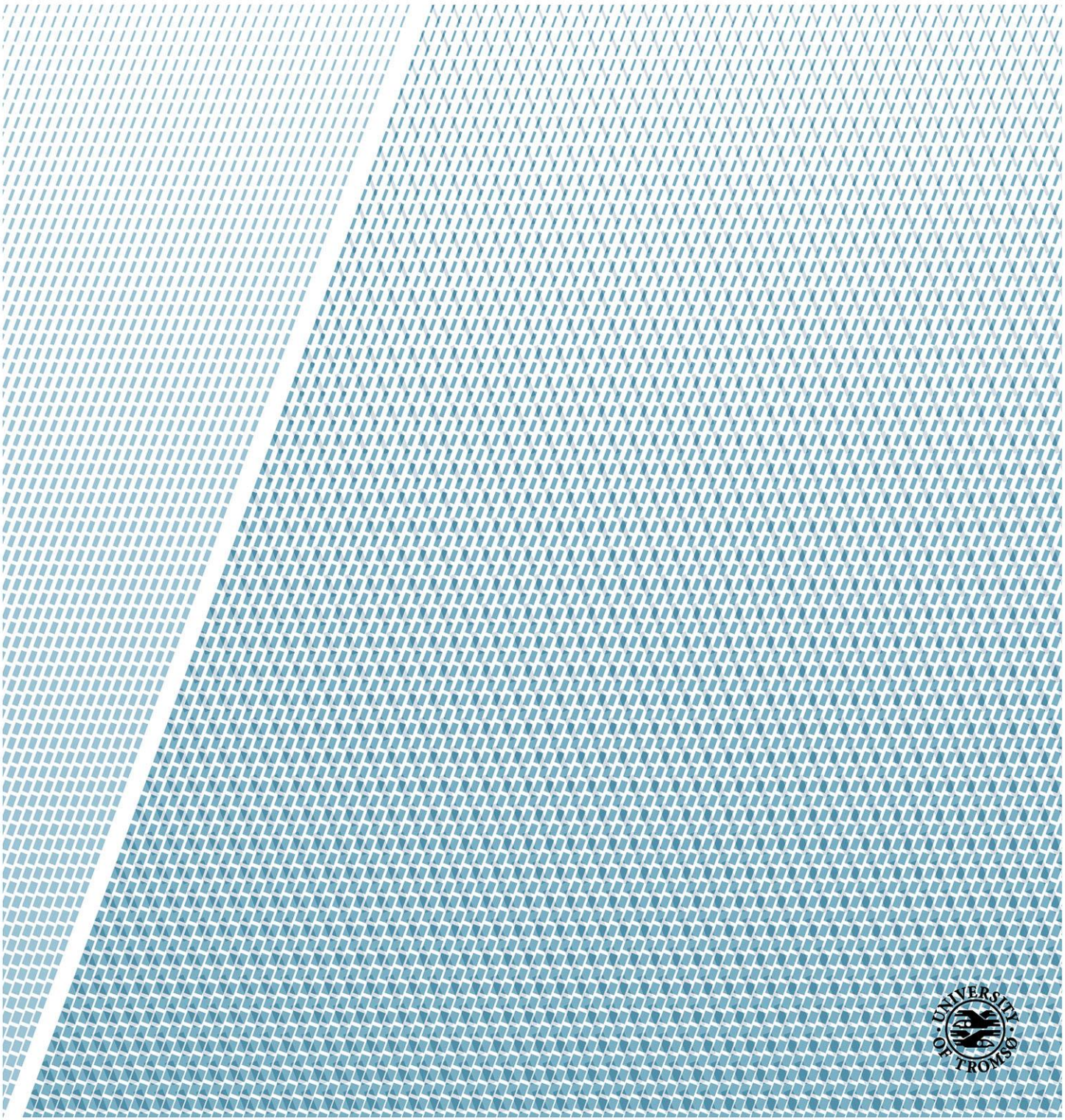
Faculty of Science and Technology
Department of Computer Science

# Controlled Sharing of Body-Sensor Data for Sports Analytics using Code Consent Capabilities

—

**Wei Zhang**

*[INF-3990] Master's Thesis in Computer Science - May 2014*



# Abstract

With the advent of body sensor technology, athletes can easily record individual physiological metrics such as heart rate, steps, and blood sugar. In parallel, there is an increasing number of web services that use the raw body-sensor data as input to sports analytics. For the individual athletes, this can yield valuable insights on their performance and suggestions on individual training programs, which consequently aid their development.

Once the data is imported into these analytics systems, the athletes are however left with little control over their data. This thesis presents code consent, a user-centric mechanism which combines informed consent and capabilities to enables athletes to share their private data in a more controllable manner. Furthermore, it gives both the athletes and analytical services the extensibility, flexibility to delegate the authority across protect domains by chaining keyed cryptographic hashes.

The action and terms of informed consent are transformed to the reference to the source code and attributes of a capability. When executing a capability, the policy of access control to the resource is enforced, and the operation to the resource is performed in OpenCPU server which is a R sandbox. With a use case, we demonstrate now a user is able to share with others a graph of his aggregated data by delegating a capability. This paper details the implementation of constructing a code consent capability, and verification, delegation, execution of a capability. The security of the prototype is also discussed when users revokes capabilities. In the prototype implementation, we also evaluate the end-to-end latency of executing a capability, which includes the time of verifying the signature, the time of executing the program code, as well as downloading the output file. The analysis of the performance guides us to investigate the optimization of our prototype such as capability cache and function chaining.

# Acknowledgements

I would like to thank my Åge Kvalnes for supervising my thesis. I would also like to thank Håvard Johansen getting me involved in a paper [1] as a second author. The experience of doing research and scientific paper writing was of great value. Thank you for the meticulous guidance and providing numerous constructive comments and detailed feedback for improving the quality of the thesis. After the discussion with you, I always get some good ideas that would speed-up and improve my work a lot. I wish I could co-operate more and learn more from you in the future.

In addition, I would like to thank Professor Dag Johansen for letting me study in the iAD group. I would like to thank Joseph Hurley for helping me set up the experiment environment. You are always kind to help me fix some implementation issues and review my thesis. Thanks to Erlend Graff, Kristian Elsebø, Einar Holsbø, Magnus Stenhaug as well as Bjørn Fjukstad for developing and sharing the latex thesis template with me. I would like to thank the Department of Computer Science, University of Tromsø, for hosting me during my master study.

Last, special thanks go to my wife Lu Li for continually encouraging me. In some sense, my master study could have not happened without your strong support. I am forever grateful.

You have your share in this work. I would not have made it without all of you!

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Capturing and recording athletes' physiological metrics through sensors is becoming increasingly prevalent. Physiological data can be obtained through different body sensors and even mobile applications. For instance, athletes can obtain their heart rate by wearing a Bluetooth enabled chest belt like the Zephyr HxM[1] and they can record their body weight through a WiFi enabled scale like the Withings Smard Body Analyser [2]. They can also get the Global Positioning System (GPS) distance data through many mobile applications like RunKeeper [2], Nike+ Running [3]. Professional sports are in particularly embracing big-data analytics using a wide-range of athletes' physical data as input, producing many types of personal and team statistics. With high level information, coaches are able to find potential performance problems from large volume of athletes' raw data and to look deep into metrics to make adjustments so that the training plan can be better tailored to each athlete.

A wide range of body-sensor data is leveraged to provide both the coach and the athlete with more accurate and objective physical development information of the athlete. For example, Tromsø Idrettslag (TIL), a Norwegian professional soccer club, has used Bagadus [4], which is a prototype of sports analytics application, to quantify both objective performance metrics to aid the development of athletes. The system uses ZXY Sports Tracking (ZXY) to

1. http://www.zehpyr.com
2. http://www.withings.com

capture a player's position, step frequency as well as heart rate. With this tracking information and the videos captured by a camera array, the coach is able to annotate and playout a particular player's video stream. At the core of this technology platform is the ZXY system, a proprietary body-area radio-based sensor network that provides raw, physical data from individual athletes to a central in-house database [5]. In addition, each player in TIL now is wearing a Fitbit Flex Wristband to capture the sleep data to know their quality of sleep.

There are also many free online Analytical Service (AS), such as RunKeeper which includes mobile application and cloud-based back-end. RunKeeper mobile application is a data capture client, which uses the sensors built in smartphones and records personal telemetry, then uploads the data automatically to the back-end of the service. The cloud-based service stores users' telemetry data and analyzes it. The calculated calories burned and distance will tell athletes the performance of the exercise. Therefore, the analytics feedback gives athletes a greater level of their performance.

On one hand, with the data captured from sensors in real time, and the sports ASs which do analytics on the data, both athletes and coaches can look deep into metrics to guide for future better training. On the other hand, since body-sensor data is private and personal data, athletes do not have complete control on their body-sensor data when the data is imported to ASs. There is no way for athletes to choose which part of data can be shared. Currently all the athlete's body-sensor data is exposed to the corresponding ASs. Plus, athletes have to rely on the services protecting their highly sensitive data. For instance, when a user installs RunKeeper mobile application on his mobile phone, every time he uses RunKeeper to track his running, all the data is uploaded to the back-end service of RunKeeper. Moreover, many third-party applications can be connected via OAuth protocol so that the user's private data, which is his body-sensor data, may be acquired by third-party applications as long as these third-parties get the tokens from RunKeeper.

## 1.1 Girji

In order to build a bridge between sports ASs and body-sensor data, Girji aims to provide a computation environment for supporting a wide rage of ASs to perform big-data analytics on body-sensor data. Girji is a computation environment that is used to host atheltes' body-sensor data so that various types of sports analytics operations can be performed collaboratively on the data. While public or proprietary services host and store athletes' data at present, these services may become bankrupt and be shut down. Girji's long-

term goal is to store and host athletes' life-time data so that the data is always available to public sectors or research institutes for analytics. An analytical service (i.e., AS) is a computational process to get insights from input data set. For example, RunKeeper web service is an AS as RunKeeper quantifies a user's performance by computing the user's positional data and calories.

## 1.2 Problem Definition

Although the emerging health data ecosystem has great potential for both users and organizations, it also poses a risk for users losing control of their private data. Existing mechanisms for access control based on service-side Access Control Lists (ACLs) are just not well suited to control data flow in this type of computational environment. In addition, the common approach to sharing one's private data with others is either by surrendering his credentials, or by copying the data and sending it. Both means are cumbersome in that the first approach gives others more access rights than they are supposed to have, and the recipient has to request the data again when it changes. Those inefficient and insecure way of sharing data hampers the collaboration among researchers in academia and practitioners in industry. This thesis shall therefore focus on mechanisms for user-centric control of personal data when uploaded and stored at health related services like RunKeeper and Withings. The goal is to develop a prototype system or mechanism that enables the users to share their data in a more confined way and to control how it flows between services. Open systems and initiatives like Ohmage and Open mHealth[3] should in particular be considered in this context. The constructed system should be evaluated in a scientific context.

## 1.3 Motivation

Users want to be able to have full control of their body-sensor data, even though their data scatters around different source services. Meanwhile, users also want to get insights of their performance by sharing their data with sports ASs in a more fine-grained manner. They need a system that is able to provide a user-centric way for users to fully control their data, to easily grant authority, and also to make it possible for sports ASs to access users' data which is authorized by the athlete.

This thesis shall design an infrastructure for retrieving users' data which is

3. http://www.openmhealth.org/

captured by different body sensors, and storing securely in the infrastructure. When the body-sensor data is hosted in the infrastructure, users are able to selectively share their private data with analytical services so that ASs can do analytics on their data. In addition to providing insightful information to users, ASs should also be able to share the result of the computation to other subjects, for instance researchers, engineers, or even end-users from another organization. While the objective facilitates collaboration, it shall not make users' sensitive information leaked out. The sharing should be confined so that ASs are not able to do more than what they are granted. In a sentence, the motivation of this thesis is to develop a mechanism to make sharing private data more controllable without giving up security.

## 1.4 Assumptions, Scope and Limitations

In this thesis, we assume that Girji is completely trusted so that Girji system itself will not intentionally disclose athletes' body-sensor data after the data is acquired from source services. We also assume that the result data must also be processed in Girji in that after executing a capability, the result data yielded from the operations can be taken out of Girji which can lead to the leakage of information. Therefore, all the raw data, and the data resulted from ASs' analytics operations, must be kept in Girji.

It is necessary to assume that the capabilities, which a user possesses, are kept securely, otherwise some other principals can have the capabilities that he is not allowed to obtain. Furthermore, the principals who gets hold of the capabilities can not only access the result data of the capabilities but also process the result data by adding operations to the capabilities and executing them. In addition, we assume that the state-of-the-art public key certificate mechanism is deployed to identify legitimate users so that we will not focus on the user authentication. When a principal presents a capability to Girji, it means that the principal has been authenticated successfully. Lastly, since the core of this thesis is to investigate how to enable controlled sharing while not giving up security, we focus on the authorization mechanism rather than the network security. Thus, Denial-Of-Service (DoS) attacks are outside of the scope of this thesis.

## 1.5 Methodology

According to the final report of the ACM Task Force on the Core of Computer Science [6], the discipline of computing is divided into the three following

paradigms:

- *Theory* is rooted in mathematics and is followed in the development of a valid and coherent theory. The steps include: characterizing objects of study, hypothesizing possible relationships among them, determining if the relationships are true, and interpreting results.

- *Abstraction (Modelling)* is rooted in the experimental scientific method, which involves the formulation of an hypothesis, model construction, prediction, data collection, and results analysing.

- *Design* is rooted in engineering, which consist of requirements statement, specifications, design, implementation and test.

The thesis is to demonstrate a proof of the concept, which addresses the problem described in the problem definition. A prototype is built to validate the design. In addition, the prototype is also evaluated to show its viability.

## 1.6 Context

This thesis is part of the information Access Disruption (iAD) centre for research. The iAD Centre targets core research for next generation precision, analytics and scale in the information access domain. Partially funded by the Research Council of Norway as a Centre for Research-based Innovation (SFI), iAD is directed by Microsoft Development Center (Norway) in collaboration with Accenture, Cornell University, University College Dublin, Dublin City University, BI Norwegian School of Management and the universities in Tromsø (UiT), Trondheim (NTNU) and Oslo (UiO).

## 1.7 Outline

The rest of the thesis is organized as follows. Chapter 2 provides the overview of the background of body sensors, sports analytics, informed consent, access control mechanisms and related work on capabilities. We detail the architecture of Girji in Chapter 3 and outline Girji's requirements in the context of privacy. Chapter 4 presents the approach to sharing user's private data in a controllable and flexible manner. In addition, we also describe the implementation of code consent capabilities. Chapter 5 evaluates both the security and the performance of our prototype with a case study. We conclude and discuss future work in Chapter 6.

# 2 Background

From academia and industry, scientists, researchers and engineers are collaborating by sharing computation outcomes or dataset resources. In this manner, the intermediate results or analysis yielded from a few hours or even days of computations can be used directly by other users. Consequently, users can learn more, innovate more together. For those valuable resources or computation results, researchers prefer to share them securely.

## 2.1 Body Sensor

In the consumer market, wearable body sensors are getting increasingly popular. With the sensing unit in the sensors, end-user's physiological data is captured, recorded. After that, the sensed data is streamed over wireless network to the information system that provides analytics. End-users are able to get wireless access to their physiological data through the body sensor, which operates as an interface between end-user and analytics systems. Body sensors can deliver important, real-time physiological information to end-users. Generally Bluetooth is used to interconnect the smart phone and the sensor. With the application installed on the smart phone, body sensors can operate synergistically with smart phones. The monitored data is uploaded on to a remote server where analytics is performed.

## 2.2 Sports Analytics

Sports organizations are able to discover, identify, and better improve the athlete's performance by applying big-data analytics onto raw physiological data, heart rate, sleep data, etc. Valuable knowledge is gained by employing computer science, statistics, and mathematics techniques and models on a collection of large and complex data sets using massively parallel algorithm and software. Since big data provides large quantities of samples, analytics operated on them reveals hidden truth. The insights in turn are used to provide recommendation, optimization or guide decision. Many soccer clubs are embracing sports analytics. Based on statistical feedback, sports analytics is helping trainers and coaches for automate decision and better train adjustment. For example, ZXY Sports Tracking system [7] presents athlete's speed, running trail, accumulated distance, fitness graph by using a chest belt capturing position, step frequency, and heart rate. In Bagadus [4], ZXY is integrated with a camera array video capture system and an annotations system. By recording the whole game and annotating soccer event, these subsystems together enable playback of a specific player and performance review.

## 2.3 Access Control

Authorization determines a principal's access rights to an object. In addition, the authority can be shared with and delegated to other principals over network, or even across different administrative domains. The access rights of principals for each object can be represented by an access control matrix [8]. Every time a principal requests to access an object, the authorization is performed by looking up the principal's access rights in the access control matrix, of which an example is depicted in Table 2.1. For instance, $USER\_A$ created the file $A.C$, so that this user has the right of *owner* (i.e., `O`), as well as *read* (i.e., `R`) and *write* (i.e., `W`). However, none of the users are entitled to execute (i.e., `X`) the file $A.C$. In practice, ACLs and capabilities are two different kinds of access control matrix's implementation. ACLs are the column-wised implementation. Each object has an ACL which lists all the authorized users along with their access rights. Capabilities correspond to rows of the access control matrix. A capability is an unforgeable digital token, ticket, or key that gives the possessor permission to access an object [9]. In a capability, there are only two items of information: a *unique object identifier* and *access rights*. There is no user identity in the capability, which means the holder, whoever he is, of the capability is permitted to perform the operations listed in the capability. Deploying ACLs for authorization in distributed systems is cumbersome. Each ACL is associated with an object. If a principal wants to review all the access rights he has, it is necessary to examine the ACL of each object. If a principal

| | A.C | TEMP | B.SH | HELP.TXT | PRINTER |
|---|---|---|---|---|---|
| USER_A | ORW | ORW | ORWX | R | W |
| USER_B | R | - | - | R | W |
| USER_C | RW | - | RW | R | W |

**Table 2.1:** Access Control Matrix example

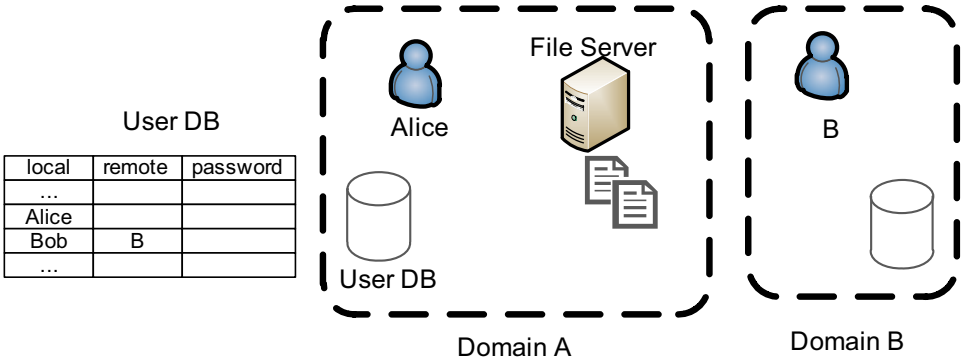



**Figure 2.1:** File sharing across domains

wants to delegate his access rights to other principals, all the ACLs, which correspond the accessed objects, have to be modified. This inconvenience may incur administrative overhead or high latency. In addition, the principal must be authenticated every time before looking up an ACL even though principal's identity does not change so much. This extra unnecessary step is inefficient. By contrast, capabilities do not require explicit authentication. Moreover, revoking a principal can be painful. All the ACLs, which include the principal, have to be updated. When authorizing across distinct administrative domains using ACLs, either a proxy or an exchange of principal account information is needed. For example, in Figure 2.1, if $Alice$ wants to grant user $B$ access to $Alice$'s files, $B$ must have a local account ($Bob$) in domain $A$'s user DB. By this mapping between the remote user identity and associated local account, user $B$ is able to access $Alice$'s files in the file server. As the size of foreign users grows large, it becomes increasingly difficult to manage the mappings. In addition, some accounts may be used for only a few times, which wastes significantly number of allocated resources.

### 2.3.1 Capabilities

Capabilities are a dual approach to ACLs [10]. In a capability-based system, it is easy to review all the access right a principal has by simply examining the principal's capability list. When a principal issues a request to the object, because capabilities are subject-based, the service provider is not interested

in if the client is known to it, the service provider needs only to check if the capability is valid and whether the requested operation is listed in the capability. While each principal carries a certificate, the principal may have a few number of capabilities. When the principal wants to request an object, he hands over his certificate to the service provider. The certificate includes not only authentication information, but also authority information such as user roles or capabilities. In addition, a capability allows the principal to loan or delegate capabilities to other principals, which is impossible in ACLs. To assure transferring rights securely, extra security measures need to be taken. To guarantee the certificate is genuine and has not been tempered with, it should be protected by means of a digital signature.

One possible access right to an object is transfer or delegation. A principal having this right can pass some types of access rights in capabilities to other principals. For instance, in Table 2.1, $USER\_A$ is the owner of $b.sh$, thus he can then delegate *read* and *write* operations by issuing a capability, which embeds object identifier and operations (e.g., read, write), to $USER\_C$. An important advantage of capabilities over ACLs is that capabilities naturally support the property of least privilege in that in ACLs the principal is able to do anything more that what he means [11]. In addition, in the distributed systems where there are a set of administrative domains, capabilities can be reused and transferred among principals, which makes them suitable for authorization across organizational boundaries [12].

### 2.3.2 *Codecaps*

A *codecap* (code capability) is a novel type of capability. With other capability based mechanisms, there is a predetermined collection of rights that can be turned on or off. By contrast, in *codecap*, the set of rights is not predefined, but can be evolved as needed. It contains embedded code which can be executed to check if the entity has rightful access to the resource. For instance, we can create a time-range right function using JavaScript, defining that the service is available only from 8:00 AM to 5:00 PM no matter who you are.

## 2.4 Informed Consent

*Informed consent* is an individual's autonomous authorization of a medical intervention or of participation in research [13]. Physicians or researchers must obtain the informed consent from the patients or subjects prior to performing any operations. In the consent, the providers (physicians or researchers) have obligation to tell the subject the procedure of the participation, the po-

tential risks, and benefits of the subject. The subject should not be deceived or coerced, which means the subject has adequately comprehended the consent form. Then he/she intentionally signs the consent. In the health care context, since athletes' body-sensor data is regarded as Electronic Health Records (EHRS), they have the right to be informed any systems that collect, store, process, stimulate these records, as well as the purpose of the research. Physicians/researchers must carry the responsibilities to conduct safe practice even though patients consent to donate their data. Patients still retain the right to file a lawsuit if physicians/researchers conduct a faulty intervention. In the health case context, athletes' body-sensor data is regarded as EHRS, and ASs are regarded as research practitioners. Thus, if any research is to be conducted on the athlete, an informed consent must be signed between the research practitioner and the athlete. The informed consent is legally effective so that if the research practitioner release the data against the athlete's will, the research practitioner is liable to prosecution. There are five components in the consent [14]:

1. Competence. The subject is capable of making decision. Subjects who are mentally retarded or receiving mental treatment are not considered competence.

2. Disclosure. The consent provider should make the subject be aware of that what type of his/her information will be disclosed, how long the information will be retained.

3. Understanding. The consent provider should explain both the risks and benefits of the participation, and let him/her know the discomfort and side effect. It is free of right for the subject to withdraw the consent.

4. Voluntariness. The subject's participation and information authorization is made intentionally by the subject.

5. Consent. The subject decides to participate the intervention, voluntarily authorize some personal information to be disclosed.

If we would like players from Tromsø Idrettslag (TIL) to donate their positional data for research. An example of informed consent is shown in Appendix A

## 2.5 Open mHealth

Open mHealth [15] is an open software architecture, which collects data by either mobile applications or on-board wearables, processes a wide range of data, and displays meaningful insights from the data. Open mHealth is designed to develop more open and modular tools to manage health. One of the key design goals is that the modules in the architecture must have standardize application programming interfaces (APIs) so that different health measures can be easily integrated and combined to provide more accurate understanding. There are three module units in the architectural abstractions for Open mHealth:

1. DSU, namely Data Storage Units, which provides a series of APIs to access data, authenticate. In order for the existing data silos to integrate with data units, a DSU poses a simple specification onto data stores. In addition, data is defined in terms of Schema ID so that any complex data structure can be referenced by a simple Schema ID. In this manner, different data structures of data silos can be accessed under the Open mHealth DSU specification.

2. DPU for Data Processing Units. A DPU is a stateless, Hypertext Transfer Protocol (HTTP) based processing module to make sense of the data. Since the data structure in DSU is represent using JSON, DPU processes the JSON data and provides open APIs for accessing.

3. DVU: Data Visualization Units. A DVU takes the data either from DPU or directly from DSU and makes the data visualized in a readable and meaningful way.

Ohmage [16] is an open-source mobile data collecting platform, which is Open mHealth specification compliant. It pushed inquiry-based surveys to end-users' mobile phones and captures, stores, analyzes and visualizes data from feedback of the surveys and the passive data such as geographical data and time. The feedback collected by Ohmage is sent back to help the doctor to see how a patient is responding and adjust the treatment. The feedback loop also helps healthy changes. Ohmage uses OpenCPU [1] to act as DVU and part of DPU. OpenCPU allows easy interpretation of insights and trends of how end-users behave. With the modular ohmage as a base stone, a lot of analysis modules and applications can be built upon it.

1. http://www.opencpu.org

# 3

# Girji's infrastructure

Girji's long-term goal is to act as a national infrastructure for access to data for research and soccer club use. The Girji infrastructure is designed to be a partially trusted broker which sits between athletes that produce body-sensor data, and analytical services (i.e., ASs) that consume the data then apply analytics on the data. Thus, there are two design goals for Girji, 1), to provide an infrastructure that is able to securely store athlete's body-sensor data which may reside in different sources; 2), to provide a controlled manner that enables both athletes and analytical services to share their authorities. An architectural overview of Girji is shown in Figure 3.1.

To meet the first design goal, Girji should acquire the data from athletes, store the data and also keep it unexposed in Girji. The reason for storing the data in Girji is to provide the athlete with an overview of his data acquired from all available sources. With this overview, the athlete is able to share more comprehensive angles of data with analytical service. Consequently, analytical service can infer more objective information about the athlete. The second consideration is to make Girji compliant with Open mHealth specifications such that athletes' data is easily integrated with external open DPUs and DVUs. In addition, since Girji is proposed as a neutral and long-standing infrastructure to store athlete data, the data should be still available even after the sensor service providers shut down. Therefore, Girji must store the data by itself. Instead of communicating with different source content providers back and forth, performance is improved when the data is stored and manipulated inside the cluster network. For example, since the online services like RunK-

eeper and Fitbit have their own cloud storage for users' data, data is uploaded to the cloud storage over wireless network (i.e., WiFi, 3G). Generally, the professional managed services of Runkeeper and Fitbit support integrations with other applications through APIs. Girji should support data acquisition via APIs provided by RunKeeper and Fitbit. After Girji extracts athletes' body-sensor data from data silos, the data should be stored in Girji's infrastructure, which is designed to be a secure container. Besides that, the athlete's data should not be accessible to other athletes. This design consideration makes the athlete more comfortable with Girji that will be storing his privacy.

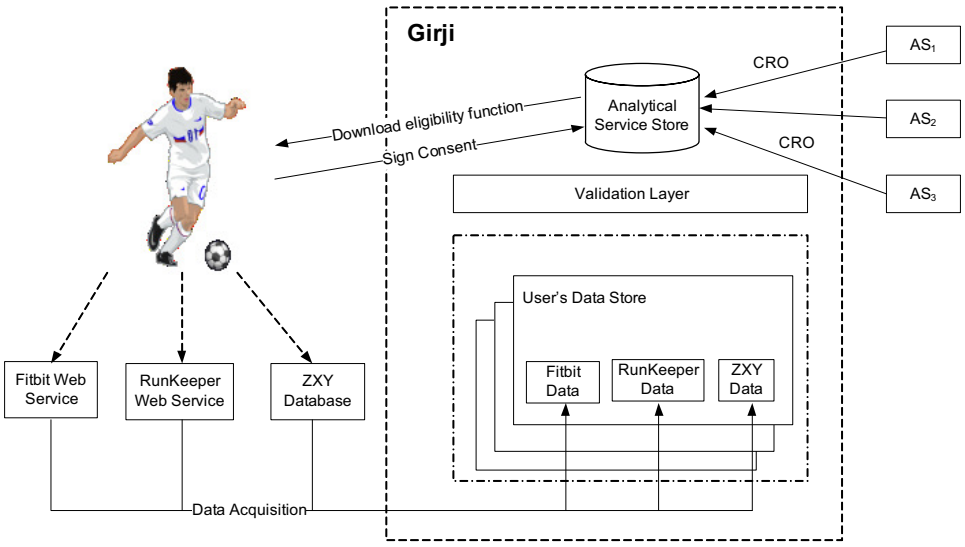



**Figure 3.1:** Overall Girji architecture

## 3.1 System Architecture

Girji architecture defines three different subsystems that are *data acquisition*, *service registration*, and *code consent management* subsystems. The sensors equipped on the user generate data which is consequently uploaded to the on-line web services like Runkeeper or proprietary in-house database. The *data acquisition* subsystem is in charge of retrieving users' body-sensor data from different data silos and securely storing the data in Girji. The reason for that is to make Girji as the only place to gather and store users' body-sensor data. It is also easy for analytical services (ASs) to access the data so that ASs need not concern about the data extraction from data silos. Each athlete's body-sensor data will be stored in Girji in a place which is completely isolated from other athletes. This data storage design adheres to the *isolation* security principle.

The information about all the analytical services is processed in the *service registration* subsystem. Prior to signing the informed consent, the athlete should first execute the eligibility function to check that if he is eligible to the service. Besides, the analytical service should also provide the source code of the operation it is going to perform. After checked to be eligible to the analytics, the athlete is able to decide whether he would consent the requested data. Each AS should retrieve only the data that is qualified by the eligibility function. After the athlete fills out the consent, and both the athlete and the analytical service sign the consent, the informed consent takes effect. Both of them will receive a hard copy of the informed consent.

The *code consent management* subsystem is used to transform the informed consent to *code consent object*, and construct the capabilities associated with the code consent object. If the athlete or the analytical service wants to delegate the capability to others, he attaches a new policy item in the policy chain and updates the signature. Further, if the athlete modifies the code consent object over time, all the capabilities created from it will be revoked and then updated to the capabilities with new policies. When the analytical service presents the capability, the capability is verified in the reference monitor of the subsystem. Likewise, capability execution is also involved in the code consent management subsystem.

In short, Girji connects different data source service providers through data acquisition subsystem and many analytical services through service registration subsystem. The analytical service's operation is performed on the athlete's data in the code consent management subsystem, in which controlled sharing is also supported.

The athletes, who generate body-sensor data by the wearing sensors, are the data producers. In other words, in Girji the athletes are the owners of their body-sensor data. The athletes decide if they agree to share some part of data, under what restrictions the data is accessed, and how long the data can be available to the AS. We introduce the informed consent for the analytical service to declare its purpose, and for the athlete to share the data with restrictions. For example, the restrictions are the accessible time period (e.g., 8:00-17:00), Time-To-Live (TTL), allow delegation, to name a few. Regarding to the AS, each AS, which wants to access the data stored in Girji, is registered in the *Analytical Service Store*. When registration, each AS should provide an *eligibility function* to filter out eligible users. For instance, a diet research project, which requires eligible participants to have a target weight of less than 55 kg over the last year, will register its eligibility function into the *analytical service store* so that athletes can download the function and execute to check if they are eligible for the diet research project before exporting their data into Girji. The *code consent management* subsystem generates an access token

| Source | Type | Data |
|---|---|---|
| FitBit Flex | Armband | Steps, calories, sleep |
| Withings WS-50 | Scales | Weight, pulse, fat-% |
| Polar RS800 | Watch | Heart rate, position, etc. |
| ZXY Sport Tracking | Belt | Position, acceleration, effort, etc. |
| MyFitnessPal App | Smartphone | Calorie and food intake |
| RunKeeper App | Smartphone | Position, calories, etc. |
| Muithu | Smartphone | Sleep quality, muscle fatigue, etc. |
| Nike+ Running App | Smartphone | Position, pedometer, calories. |

**Table 3.1:** Body-sensor data sources example

which is essentially a capability with access policy in it. The policy is specified in the informed consent which is transformed into the code consent object. After that, the access token is sent to the AS so that the AS is able to access the data which the athlete consents to donate.

## 3.2 Data model

In Girji we have adopted a simple but effective scheme that captures all the athlete's body-sensor data as a data archive $R$. The athlete's data archive comprises data sets captured from different body sensor sources $s = [s_1, s_2, \ldots, s_n]$.

$$R = \{r(s_1) \cup r(s_2) \cup \ldots \cup r(s_i)\}$$

In the equation, $r(s_i)$ is the set of records from a specific body sensor source $s_i$. Since the body sensor source may capture more than one type of body-sensor data, $r(s_i)$ consists of a collection of different types of sets of $n(s_i)$ records from the same body sensor source $s_i$.

$$r(s_i) = [r_1(s_i), r_2(s_i), \ldots, r_n(s_i)]$$

For instance, the Fitbit Flex armband is able to capture the heart rate, sleeping data, apart from step count. Therefore, the set of records $r(s_i)$ (i.e., $r(fitbit)$) is composed of $r_1(s_1)$ (i.e., heart rate records), $r_2(s_2)$ (i.e., sleeping data records), and $r_3(s_i)$ (i.e., steps records). An example of different body-sensor data sources and the types of data each source can capture is given in Table 3.1.

A record captured at a timestamp is the smallest unit of data in Girji. No matter it is captured by a scale, a smartphone app, or an armband, each record $r_s$ is represented by a monotonically increasing timestamp $t$, a time

offset $\delta$ and a vector of values $\vec{(v)}$ captured by the body sensor.

$$r_i(s) = [t_i, \delta_i, \vec{v}_i], \quad i = 1, 2, \ldots, n$$

Here $t$ denotes the timestamp when the sample is recorded (e.g., Unix time). The source $s = (type, device)$ identifies the type of record, like "position" or "pulse" in combination with what device that generated the data, like `zxy.belt` or `RunKeeper.app`. By explicitly stating device names, Girji can support multiple devices that provide similar type of data. For instance, both the RunKeeper app and the Polar watch provide positional data. To describe device names, we have adopted a dot-separated hierarchical notations, similar to Universal Resource Locations (URLs), which enables efficient management of name spaces with many vendors and devices. The vector $\vec{v} = [v_1, \ldots, v_n]$ denotes $n$ source specific measurement values for the sample, and may contain arbitrary data like integers, text strings, or even large binary objects like images and sounds.

In the data set of records from the same body-sensor data source, not all the different types of records have the same representation scheme. For example, weight data and blood pressure data are discrete records while positional data and heart rate are continuous records which are sampled in every second or even shorter time. What's more, for the positional data records, it makes more sense to capture records in some time span. For this reason, we need to indicate the start time of the capturing and the end time. Each record $r$ include a time offset $\delta$ that indicates the timespan $[t, t + \delta]$ for when $r$ is valid. For most low-level sensor records, like positional data from a 1 Hz GPS device, $\delta = 1$ indicating that the record is valid until the next sample. A value $\delta = 0$ typically indicates that a sequence of samples are ended. This is important in order to distinguish time-spans with no samples from time-spans between two valid samples. We can also use this facility to capture high-level meta-events that relate to the users. For instance, during a soccer match, we can capture that for a given timespan the user $a$ played in TIL's game against Strømsgodset on November 3rd 2013 by adding:

$$\begin{aligned} r\left((\text{events.soccer.match}, \text{zxy.app})\right) = &[1383505200, 10800, \\ &(\text{TIL}, \text{Strømsgodset})] \end{aligned}$$

Here the date (i.e., Sun, 03 Nov 2013 19:00:00) is transformed to a Unix timestamp 1383505200 and the game was recorded for two hours which are 10800 seconds in other words.

## 3.3 Service registration subsystem

For the users, it is difficult to decide which ASs to give authorization of their data to unless each AS gives users an informed consent which describes the purpose of the AS and what types of body-sensor data the AS will be using. For the ASs, not all the users are the eligible participants for the AS's analytics. ASs should use the data only from the eligible users. The service registration subsystem is the bridge to make users and ASs know more information about each other. To gain access to the body-sensor data, each AS registers a Consent Request Object (CRO), containing an *informed consent* and an *eligibility function*, with Girji's AS registry.

An *informed consent* which is an authorization document permitting the disclosure of protected health information. In order to perform analytics on user's body-sensor data, each AS has to register an informed consent onto Girji so that the eligible users are able to look at the terms and decide if they are willing to share their data. After each AS registers its informed consent in the service registration subsystem of Girji, users are able to see what each AS is about and what types of body-sensor data they are likely to authorize if they are eligible for the AS. In the informed consent, the AS describes the purpose of the analytics, and perhaps the risk and benefits of the analytics. The AS also requests the types of body-sensor data it is going to use. After reading the informed consent, the user has in his mind what the AS is about, then he can decide whether he is going to give authorization of access of the data requested by the AS. In order to enable the user to give the authorization in a more fine-grained and more controllable way, there is policy that the user can specify in the informed consent. Rather than only including the constraints in the policy, the reference of the operation is also incorporated in the policy. As we know, the operation is performed in terms of the source code. The reference of the operation is a soft link to the source code. The constraints are the predicates which will be evaluated before granting the operation. If all the predicates are evaluated to be true by the reference monitor of the code consent management subsystem, the operation is performed. The constraints are as follows:

- *access_period*, during which time the requested data is accessible to the AS.
- *allow_delegation*, which indicates whether the AS is allowed to delegate its capability to other ASs.
- *Time-To-Live (TTL)*, which is the retention time of the data.

An *eligibility function* which is used to filter out the users that meet the re-

quirements of the AS description of the service. Checking the user's eligibility is the prerequisite of the analytics. Before executing the eligibility function, it is not yet clear whether the user is eligible for the AS. The user's privacy is leaked out if the user uploads his sensitive information up to the AS to execute the function. In order to better protect user privacy, instead of uploading the user data to the AS to execute the *eligibility function*, the eligibility function is shipped to the user's side to be executed. For instance, the function is downloaded to the user's smartphone or computer to check the user's eligibility so that the user's data is not disclosed to the AS. No data is allowed to be transferred back to the AS. The execution result of the eligibility function will tell the user if the user is qualified for the AS's analytics. This is the first step for the user to participate in the AS's analytics. If the user is eligible, after he signs the informed consent, the data requested by the AS is accessible to the AS. However, the data will not be exported to the AS. The reason for that is in our design ASs are not trusted. The source code of the operation applied to the data is executed in Girji's sandbox.

When registering with the subsystem, each AS is required to upload an eligibility function and an informed consent file in which TTL, access_period, allow_delegation, data_range and code_ref fields are included. Each AS will be given a global unique id after the AS is registered in the subsystem. After registered, the data structure of analytical service registry is shown in Figure 3.2. The informed consent is stored as an XML file because it is more readable whereas the eligibility function is stored as an executable file as it will be shipped to the user's smartphone and executed. The access_period, allow_delegation and TTL fields are specified by the user who is eligible and also consents to share the data. The reason for storing them in each column is to enable the subsystem to retrieve the value directly from the registry rather than extracting from the informed consent file.

| AS_ID | consent_file | eligibility_file | code_ref | data_range | access_period | delegation |
|---|---|---|---|---|---|---|
| as_0 | xml_0 | exe_0 | https://ref0 | weight(2013) | 8:00-12:00 | true |
| as_1 | xml_1 | exe_1 | https://ref1 | running(2013.11) | 8:00-17:00 | false |
| ... | ... | ... | ... | ... | ... | ... |

**Figure 3.2:** Analytical Service Registry

## 3.4 Data acquisition subsystem

It is common that a user has several body sensor sources such as a RunKeeper app, a Fitbit armband and a ZXY belt. Meanwhile, all these sources have their own web services to help the user host and manage the body-sensor data.

Thus a user's data archive $R$ is dispersed in different data silos. Further, the body-sensor data is stored by different scheme in those data silos. The *data acquisition* subsystem is used to retrieve the data from different data silos and store it in Girji so that Girji is able to provide a uniform access scheme to ASs. In addition, each AS is relieved from interacting with different web services by various APIs. ASs can focus more on its analytics job. The body sensor store in Girji can be considered as a cache for the corresponding data silo. The AS will access only the data available in Girji. If the AS requests to access the data which is more updated than what is stored in Girji, the more updated data will be synchronized into Girji's data store by the data acquisition subsystem. After the data is acquired into the Girji infrastructure where all the ASs' code generally resides in the same network as the data, the data can be accessed very fast. There are two ways for Girji to acquire the user's data from data silos:

- The user let Girji know his credentials of the web service so that Girji can retrieve the body-sensor data from the web service, for instance, in this way Girji can retrieve the body-sensor data in the in-house database of ZXY.

- The user does not have to share his username and password with Girji if Girji is connected to the web service which supports OAuth protocol.

A user's body-sensor data is his/her sensitive data. Thus, the data should not be disclosed to ASs unless the user gives authorization. In addition, the user's data should not be accessible to other users. In Girji's design, all the data, which is from different sources, of the same user is stored in the user's *infospace* that is completely isolated from other users' storage. In the infospace, which is shown in Figure 3.3, there are many data stores and each store represents an individual type of sensing data, e.g., positional data, steps, weight, heart rate. The reason for having each store for each type of data is that the sample rate of sensing data may be significantly different. Another consideration is to better reflect the characteristics of each sensing data. For example, the weight does not change every hour whereas the positional data or the steps are sampled every ten seconds. Since the primary key is the combination of the timestamp and device, if one hour of positional data is captured and the weight value is appended to each row, then the redundant weight value is stored for 360 times. A user's infospace can be implemented as a separate database or be stored in an isolated virtual machine.

Since each piece of body-sensor data is associated with a timestamp and a source, the piece of data and the timestamp in combination with the source are stored in each row of the Girji's body sensor database. Therefore, the primary key of each table is the timestamp column together with the source

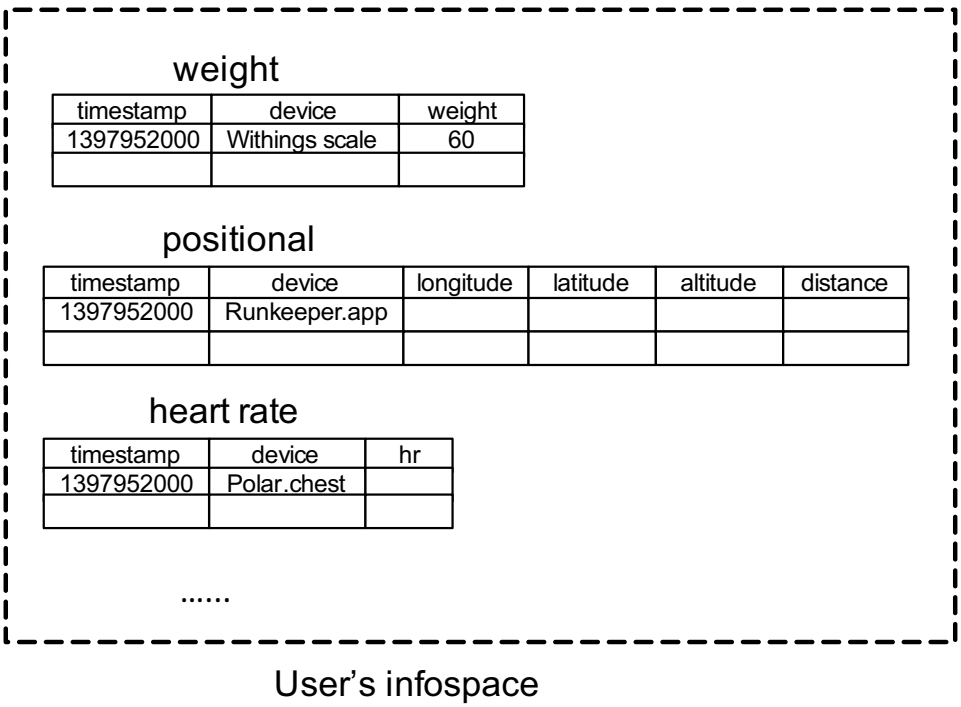



**Figure 3.3:** The design of user's *infospace*

column. Each row of record can be retrieved by SQL query so that each row is the minimal granularity in Girji. This relation schema makes more fine-grained sharing possible so that the user is able to share each row of data or even the same row of data with different policies. For the rows of data which has an obvious event (e.g., a soccer game), it is natural that all those rows of data corresponds to one code consent object. Then Girji manages those rows of data as a whole and the user authorizes either all the rows of data in a game or none of the rows. It makes sense in some cases, for example, the data of a game, the data of a workout. Yet, for the data, which has no specific event to correlate to, such as heart rate or sleep data, it is difficult to have those rows of data correspond to an event. For example, some ASs may request the user's heart rate data captured from 18:00 to 0:00 while some other ASs may request the data captured from 0:00 to 6:00. We have no idea what data the AS requests to access. In this scenario, there is no specific event to associate with the six-hour heart rate data. In order to be able to share the data, we have to deploy granular access schema.

In the database, there is only body-senor data from sources and no authorization data is stored. The authorization data is stored in another database in the *code consent management* subsystem. When acquiring the data from sources, it is natural that different sources store the data by different schemes.

In order to provide a uniform scheme to ASs, all the different fields will be appended as individual columns in each row of Girji's database. By this means, all the related information is retrieved from different sources and stored in the same row in Girji's database, which is illustrated in Figure 3.4. In the example, since both the runkeeper app and fitbit armband can record a user's running activity and they store the data in different schemas, all the column data is retrieved and stored in Girji's body sensor database so that each row of data will have not only `altitude`, `longitude`, `latitude`, and `distance` from runkeeper, but also `steps` from fitbit. This design scheme can provide as much information as possible that is associated with each row of data. When sharing, this further enables ASs to have more objective outcomes.

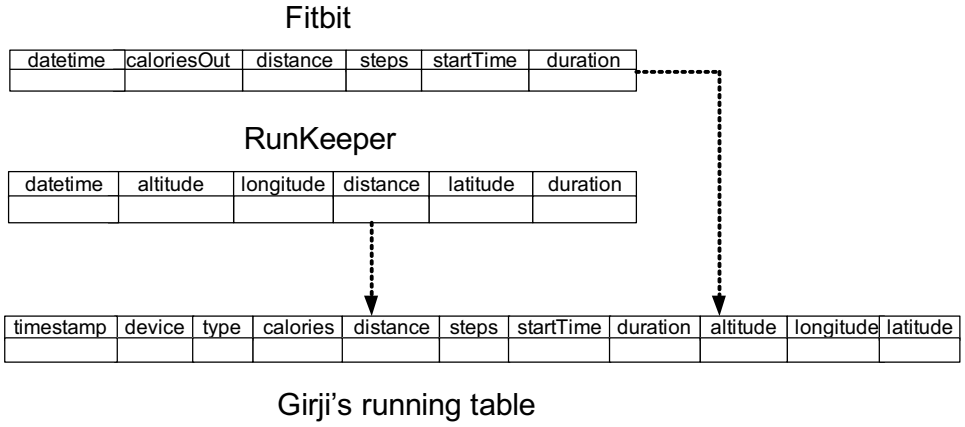



**Figure 3.4:** Data schema

In short, the data acquisition subsystem is in charge of retrieving the user's body-sensor data from different sources either via OAuth or by proprietary APIs. In the subsystem, all the data of a user will be stored in an isolated *infospace* which is not accessible to other users. The data of the same type resides in the same store in which each piece of data together with the timestamp and *device* is organized as a row of data record. This relation schema fits better the characteristics of the sensing data, such as positional data sampled at high frequency whereas weight data at low frequency. With the timestamp and device as the primary key, Girji is able to support the same type of data captured by different sensors. The relevant fields, which are returned when retrieving the same type of data from different silos, are appended to each row of the associated data store.

### 3.4.1 Data acquisition from RunKeeper

The Health Graph APIs [17] provided by RunKeeper enable Girji to access body-sensor data like fitness activities, weight, and sleep measurements stored within their services. Access to RunKeeper's Health Graph API is managed through the OAuth 2.0 authentication protocol [18], which is commonly used by other Internet services like Facebook, Twitter, and Google to authenticate and authorize third-party applications. The benefit of using OAuth is that users can give third-party applications like Girji access to their RunKeeper account without having to share their credentials with the applications.

To access data in RunKeeper, we therefore first need to register Girji as an application in the Health Graph system. After providing RunKeeper with the name of our application and the URL of our web-site, Girji will receive an identity token (e.g., `d28****e91`) and a secret (e.g., `ec1****24f`). When the user wants to access his health graph through Girji using the OAuth2.0 protocol, the following steps will be executed:

1. The user is directed to the Health Graph API authorization URL, with the request parameters: identity token and Girji's URL. For example, in our case the HTTP Get request is https://runkeeper.com/apps/authorize, with parameters

   ```
   client_id=d28****e91,
   response_type=code,
   redirect_uri=http://girji.no/main
   ```

2. The user is prompted to input his RunKeeper account name and password. The account and password are not revealed to Girji. After the user is authenticated by RunKeeper, he is prompted to accept that Girji is allowed to access his health graph data. If the user permits this, the Health Graph API will redirect him to the `redirect_uri`, which is http://girji.no/main, and one-time code `53c****977` for Girji to get the access_token afterwards.

3. Girji sends a POST request to https://runkeeper.com/apps/token with the request parameters one-time `code`, which was returned in previous step, `grant_type`, `client_id`, `client_secret`, and `redirect_uri`. An example of POST request is https://runkeeper.com/apps/token, with request parameters

   ```
   grant_type=authorization_code,
   code=53c****977,
   client_id=d28****91,
   ```

```
client_secret=ec1****24f,
redirect_uri=http://girji.no/main
```

4. An access token will be included in a response to Girji. This access token is uniquely associated with this specific user. The token should be included to each request made by Girji to access the user's health graph.

After Girji gets the access token for the user's health graph, Girji is able to retrieve the users fitness activity data by sending a HTTP Get request to `http://api.runkeeper.com/fitnessActivities`. The RunKeeper web service will reply with `200 OK` together with a list of `FitnessActivityFeed` in JSON format. Each item of FitnessActivityFeed represents each fitness activity's summary including start time, fitness type, distance and the uri of the activity which the value of altitude, latitude, longitude of each path point can be retrieved. The details of an individual fitness activity can be acquired by sending another HTTP GET request by specifying the uri of the activity like `/fitnessActivities/40`. When the user logs into Girji's portal and clicks `Connect to RunKeeper`, a summary of the user's fitness activity history will be shown. An example of this table is shown in Figure 3.5. Meanwhile, a `fitness.csv` file of the user is created and stored in this user's directory.

User's RunKeeper Data

| Date | Type | Distance(km) | Duration | Calories |
|---|---|---|---|---|
| Tue, 18 Mar 2014 21:51:48 | Cross-Country Skiing | 8.01 | 1:15:17 | 647 |
| Sat, 15 Mar 2014 18:02:22 | Running | 5.55 | 0:31:41 | 357 |
| Sat, 15 Mar 2014 12:58:41 | Walking | 2.15 | 0:33:06 | 121 |
| Sun, 23 Feb 2014 12:50:12 | Running | 2.29 | 0:19:03 | 147 |
| Wed, 19 Feb 2014 17:46:19 | Running | 5.81 | 0:40:15 | 412 |
| Sun, 16 Feb 2014 14:04:22 | Downhill Skiing | 21.36 | 2:25:06 | 373 |
| Sat, 8 Feb 2014 12:56:04 | Walking | 2.69 | 0:40:40 | 198 |
| Sat, 5 Oct 2013 20:31:59 | Walking | 2.26 | 0:30:18 | 107 |
| Sun, 29 Sep 2013 12:58:50 | Walking | 1.77 | 0:29:25 | 98 |
| Sat, 21 Sep 2013 17:49:32 | Running | 3.56 | 0:25:21 | 242 |
| Sat, 7 Sep 2013 17:55:30 | Running | 7.48 | 0:52:31 | 517 |
| Sun, 1 Sep 2013 16:26:09 | Running | 3.49 | 0:27:16 | 228 |

**Figure 3.5:** An example of user's fitness activity history

## 3.5 Requirements of Access Control

One of Girji's principal objectives is to enable users to selectively share their body-sensor data with proper privilege. There are functional properties of access control specific to private data sharing in Girji. A user should be able to easily share his private data to others without the intervention of system administrator as in a large-scale system it is cumbersome for administrator to be involved in every process of sharing. Apart from the ones registered in Girji, one should be able to share data with users outside Girji, which means sharing across administrative domain. In addition to that, Girji also encourages more collaboration between ASs by sharing their interesting analytics results in a confined manner. In addition to sharing data, a user should also be able to delegate access authority to other users in and/or outside Girji. Since a user's body-sensor data may change dynamically as he has more activities, it is desirable that other users are able to get the access right to his updated data. Lastly, since a user can give access rights to others, we should also be able to revoke some access rights.

While athletes are the body-sensor data producers, ASs are the data consumers. Operations on the athlete's data should be granted by the athlete. Otherwise the operations will be rejected by Girji if the athlete does not give permission to the AS. Generally, an AS could be either a scientist, a researcher, or a private company running analytics. Combining different sources of data and taking it all in the AS analytics can yield more insights. However, athletes' data might be leaked out by ASs accidentally or on purpose if ASs have access to all the data. In order to protect user privacy, the AS is allowed to access only the data which is needed. This requirement corresponds to the principal of least privilege in information security.

The requirement of fine granularity applies to two aspects i.e., shared data and access rights. The first aspect is that the access control mechanism should enable athlete to be able to specify smaller data items to be shared with others. Yet, an extremely fine-grained authorization can lead to very much administrator management and low performance, e.g., specifying access control based on each millisecond makes no sense when access control based on each second or each day is desired. Therefore, the access control mechanism should support a flexible level of access granularity of operations and access context. The other aspect is that principals should be able to confine the access rights when delegating. The principal should be able to delegate only a subset of his access rights to the recipient principal who can access only the resource specified by the grantor.

Autonomous delegation between principals in different ASs is requiredFor ASs, there are distinct administrative domains. Some researchers might want

to share their extemporaneous experiments to some ones in different domains without much interference of administrators. In other words, either athletes or ASs should be able to delegate their authorities with minimal efforts, even across domains. No administrator involvement should be required.

In order to mitigate the privacy risks, the mechanism should provide means for athletes to revoke access rights of the AS that released the data. Since ASs are not trusted, they could accidentally or intentionally leak out the data. There is also some other reasons athlete should revoke some access rights. For example, one should revoke all the access rights of his previous club after he moves to another club.

The second requirement is that Girji should enable the athlete to set the policy in terms of the type of the data, the time of retention and delegation. Moreover, the policy is also enforced in Girji to realize the fine-grained and user-controllable data access to ASs. The access request from the AS should always be checked by Girji based on the policy set by the owner of the data. Last, in case of emergency situations, the athlete should be able to revoke the AS access to his data at any time. Once accepting the AS's access right, the athlete's sensitive information is always available to the AS. It is possible that the athlete's identity might be revealed due to the development of the AS's algorithm. Therefore, there should be a mechanism to make the athlete be able to revoke the access anytime he wants to do so.

Based on different athletes' privacy needs, the prototype should support policies of different privacy levels. The athlete should know clearly what type of data is stored in which system. When the sports analytics system is passing the athlete's data to a third-party sports analytics system, the athlete should be notified to determine interactively whether to grant access to the data or not. When the timetolive time expires, the analytical service is no longer allowed to access the data while the data is still stored in Girji. The requirements, which the access control mechanism should satisfy, can be summarized as follows:

- More fine-grained data sharing without administrator intervention.
- Least privilege of access rights.
- Easy delegation and revocation across administrative domain.
- Mechanisms to remove data automatically from ASs.
- User-data traceability.

# 4

# Code Consent Capabilities

Based on access-control requirements from Chapter 3, this Chapter will introduce and describe our approach to satisfy these requirements in the context of Girji. After user's data is retrieved and stored in Girji, the data is ready to be shared to ASs. Since the data is highly personal, the data should be shared in a restricted manner. Moreover, when ASs yield results from the raw data and want to share them, some access control policy should also be enforced because the results are derived from user's data and the user is the owner of his data. Therefore, the access policies specified by the user apply not only the raw data but also all the derived results. Meanwhile, the access policies specified by ASs should not be conflict with the policies specified by the user. Another consideration of access control is that Girji allows users and ASs to share the data to others who may not be register in Girji. For sharing data across administrative domains, capabilities mechanism is chosen over ACLs by the following reasons:

- Large number of subjects and objects may be frequently added or removed, which makes changes in ACLs is inefficient;
- When an object is created, few subjects have access to the new object. Thus most entries of ACLs are empty.
- Since access right delegation is very common in Girji, it is with less difficulty to do delegation with capabilities as the delegation in ACLs will result in the scan of all ACLs.

- Within ACLs, delegation across domains will not be possible without proxies.

Furthermore, in a distributed system, it is sufficient to trust a subject with the credentials. And because the capability is sent along with the request, the time spent for accessing attributes is much less than the time for searching an ACL.

## 4.1 Design

Access control in Girji is managed using self-contained and highly expressive capabilities similar to that of the Codecaps and Macaroons [19, 20]. The *operations* and *constraints* in the capabilities are set from the content in the informed consent. In order to encourage sharing while not compromising user's private data, we deploy capabilities to entitle principals to have restricted access rights which are represented as a list of combination of operations and constraints.

After ASs are registered in Girji's *AS store*, the consent file in each CRO is listed so that eligible users can review the content of the consent file. If it turns out that the user is eligible for the AS's analytics by executing the eligibility function, the user can review the `dataRange`, `codeRef` fields to decide if he wants to share his data. The user sets some *constraints* and signs the informed consent. At the same time, a hard-copy informed consent is generated so that the paper certifies that the AS is responsible for performing analytics on the data but is not allowed to intentionally leak out the data. A code consent object is also created. The purpose of code consent object is that 1), keep in memory the representation of the informed consent; 2) user for minting the root capability. The *codeRef*, *constraints* fields in the code consent object are copied to construct a list of policy chain in the capability. After computing the signature of the policy chain, a root capability is minted and sent to the AS so that every time AS wants to perform the analytics, it simply presents the capability to Girji's reference monitor and executes it. Each capability has to go through the reference monitor. When the reference monitor receives the capability execution request, it will first verify the capability, then execute the policy chain from root to last. During the execution, the AS cannot access any resource except the result of the execution of the policy chain. Therefore, the result is returned to the AS. The overall design of code consent capabilities is shown in Figure 4.1

The first phase is to construct capabilities based on the code consent object or to delegate capabilities based on existing ones. In the *code consent manage-*

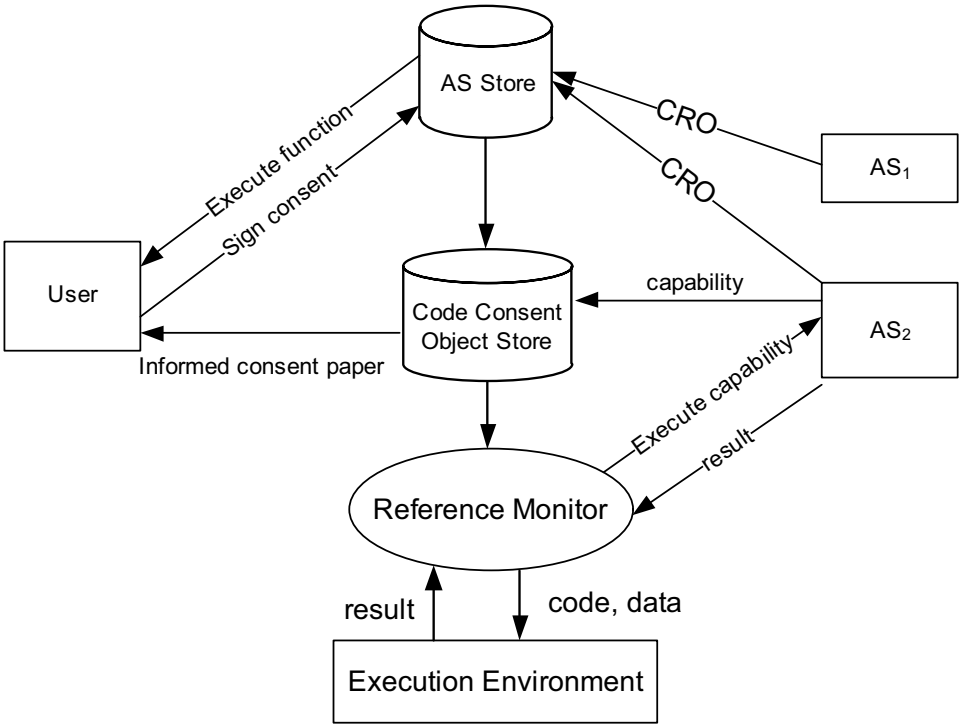



**Figure 4.1:** Overall design of code consent capabilities

*ment* subsystem, a new CCO is created every time a user signs an informed consent. After that, a capability is constructed in accordance with the code consent object, and then passed to the AS after which the AS has the user's authority to access the data specified in the `dataRange` field of the capability. However, the AS can only perform the operations specified in the *codeRef* of each policy item.

The second phase is to execute the capability so that principal is able to apply operations which are in the policy chain, and access the result. When a principal presents a capability to Girji, the *reference monitor* will be invoked to enforce the access control policy. Instead of storing the policy in a database, the policy is embedded in the capability. Therefore, policy enforcement is essentially to perform the operation on the data under some constrains. The operation, data range and constrains are all contained in the capability. After the reference monitor verifies the capability's integrity, freshness and evaluates the constrains, if all the checks and evaluation are true, the operation on the data is granted. Lastly, the operation is performed in an HTTP-interfaced sandbox (i.e., OpenCPU). In summary, the access control mechanism should implement the following functions:

- Mint the capability based on the corresponding code consent object.
- Strictly execute the capability.
- Capability delegation.
- Capability revocation.

## 4.2 Code Consent Object

A code consent object (i.e., CCO) is a data structure representing the informed consent and used to associate all the capabilities minted from the this code consent object. After the user specifies the *constraints* and signs the informed consent, a code consent object is created. The code consent object consisting of a *CROId*, a *provider*, *consenter*, *CCOId*, *revoked* as well as some fields representing the terms of the informed consent such as *dataRange*, *codeRef* and a list of *constraints*. The *CROId* field identifies the CRO which this code consent is created from. *consenter* denotes the user who has signed the informed consent while *provider* is the identification of the AS. Instead of embedding the source code, the link address of the code is incorporated in the object as the source code of the analytics may be quite large. Examples of constrains are the access time period (*accessPeriod*), whether to allow authority delegation (*allowDelegation*), how long the AS can retain the data (*TTL*), which ip address is allowed to access and so on. The value of the operation (i.e., codeRef), which will be executed to perform analytics on the range of data (i.e., dataRange), is passed from the informed consent. Both the *dataRange* and *codeRef* are unmodifiable in that the user has signed the informed consent. What the user can change over time is the list of constrains. The constrains are a number of predicates that will be evaluated to restrict operation. When sharing data to the same AS, different users have different code consent objects in that they may specify different constraints.

A CCO is a in-memory object which is managed by Girji while a capability will be sent to the AS and kept securely by the AS. Each AS registers only one CRO in Girji. However, after signing the consent files, different CCOs are created for different users in that the consenters for the same CRO are different and they may set different constraints. Consequently, capabilities, which are minted from different CCOs, are not the same. The relationship between CRO, CCO and capability is shown in Figure 4.2. An individual CCO is identified by `CCOId`. In addition, different users (i.e., `consenters`) may specify different `constraints`. The relationship between CRO, CCO and capability is shown in Figure 4.2.

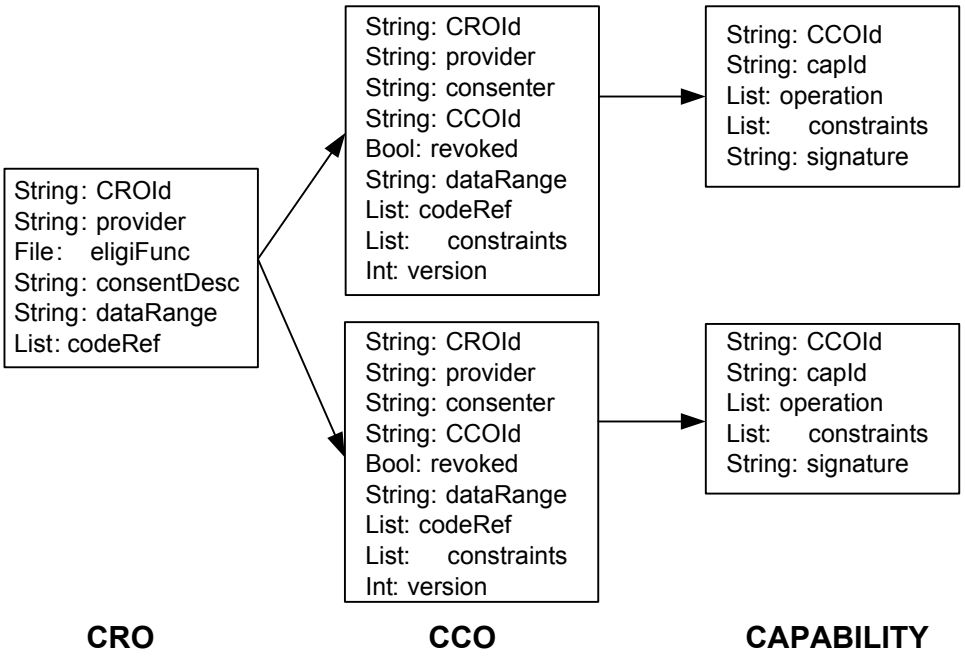



**Figure 4.2:** The relationship of CRO, CCO, and capability

Code consent object is used to construct the first capability (i.e., root capability). The capabilities, which are minted or delegated afterwards, are all associated with the code consent object. After the code consent object is created, it is the basis for constructing the root capability because the value of *codeRef* and a list of *constraints* is copied to construct the policy chain in the capability. After setting the value and computing the signature using Hash-based Message Authentication Code (HMAC), a self-contained capability for performing operations under some constraints is minted out.

In addition to construct the root capability, code consent object is also used in the execution of a capability. Every time the reference monitor verifies the capability, it will check if the *revoked* field of the code consent object with which the presented capability is associated. If it is `true`, this means the capability has been revoked so that the access is denied. Besides, since the user can modify the constrains of the code consent object as needed, the constrains in the capability might not be the latest ones. The reference monitor will compare the constrains of the capability with the ones of the code consent object. If the constrains are not identical, which means that the user has modified the constrains, the constrains of the capability will be updated to the constrains of the code consent object.

## 4.3 Code Consent Capability

A *code consent capability* is essentially a capability, which has a nonce of *capId*, a string of *CCOId*, a chain of policy items and a string of signature. The globally unique *CCOId* identifies one specific cco object stored within Girji. The signature is generated by computing the hash of the fields of the capability, and is also used as the key of the hash function for computing the signature of the delegated capability. The policy list $chain_n$ is a chain of policy items

$$chain_n = [p_0, p_1, \ldots, p_n]$$

which corresponds to a chain of combination of *operations* and *constraints*. Under each constraint, the execution environment will execute the operations from the root policy item (i.e., $p_0$) to the end policy item (i.e., $p_n$). Each policy item $p_i, i = 0, \ldots, n$ is a collection of *attributes*. A policy attribute is a name-value pair. We denote by $p_i$.`attr` the value of the attribute named "*attr*" in the policy item $p_i$. The *operation* attribute of the first policy item is set by AS and *constraints* are set by the consenter. When delegating capabilities, which means the grantor is giving the grantee some confined access right, both *operations* and *constraints* are set by the grantor.

The benefit of incorporating the signature and the chain of policies in a capability is that in a distributed system this design reduces the communication overhead between capability component and Girji. While there are one or more capability components to support legacy data sources for capabilities access control, there is only one execution environment in Girji. The decoupling of execution environment and capability components makes Girji easily scale out. If Girji wants to support another administrative domain, a capability component is added in Girji so that capabilities of accessing data on behalf of the domain's principals are created by that component. The capability component is also used to adapted to legacy data source for supporting capabilities access control. The capability component is able to mint a new capability based on the preceding capability as the signature of the preceding capability can be used as a hash key for computing the signature for the new delegated capability and the policy chain is contained in the preceding capability. A capability component manages code consent objects and constructs new capabilities by building up policy chain and computing signature. Different capability components manage not the same code consent objects and the information of ccos will be synchronized among the capability components. When a capability component constructs a new capability by delegating, the value of the signature depends on only the policy chain which is already contained in the capability. Thus, the capability component need not communicate with the execution environment. Since the signature of the capability is used as a hash key for computing the signature of the delegated

capability, the capability component simply executes HMAC function. As for minting a root capability from scratch, since the *operations* and *constraints* are all included in the code consent object, the values can also be got from the capability component instead of the execution environment. The difference between capability component and proxy is that the new capability, which is created by the component, is executed in Girji's execution environment instead of in a proxy. A new capability can be created by a proxy. However, since a proxy sits between Girji and the principal, every capability is executed in the proxy and then the proxy in turn presents the resulted capability on behalf of itself and executes it in Girji. By contrast, the capability minted by capability component can be executed directly in Girji's execution environment. The difference between proxy-based and component-based design is shown in Figure fig:proxy.

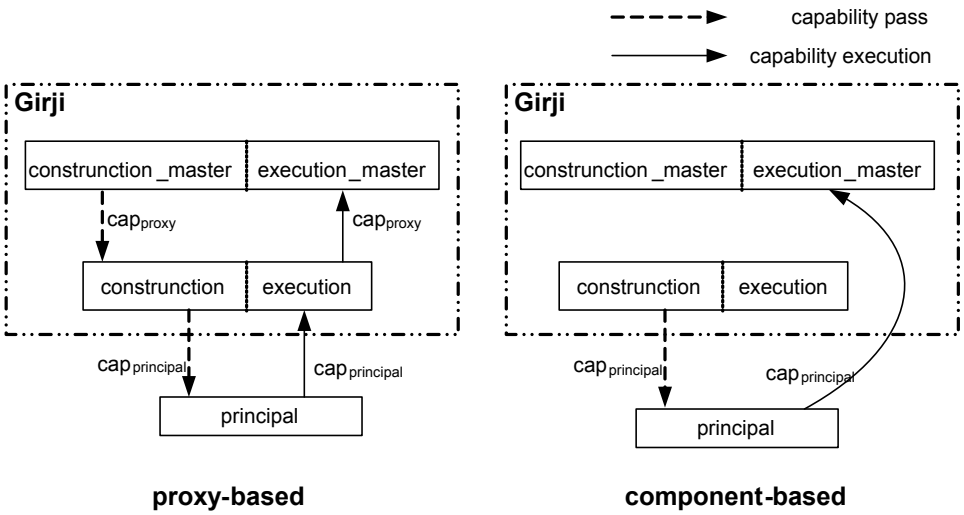



**Figure 4.3:** Different between proxy-based and component-based design

A policy item consists of a *constraint*, and an *operation* field. *Operation*, which is the code reference *codeRef* along with parameters (i.e., *param*), is what the principal can do with the input data. *constraint* is the predicate evaluated by Girji before the *operation* is performed. For instance, the restriction [8:00-17:00] means the operation in the capability can only be performed during that time period. The value of *operation* and *constraints* of the the first policy item in the policy chain is the value of that of the corresponding CCO. When a user signs the informed consent, there is one CCO generated from the content of the informed consent. However, there may be one or many code consent capabilities associated with the CCO as many capabilities may be delegated from the root capability. When a principal wants to access the data of a user, first, the principal extracts the policy chain from the capability; second, executes the operations in the chain from the root one to the last one. Since

the operation and constraint are all contained in the capability, what the principal has to present is only the capability. The primary data-access API call provided by the Girji has the form

$$principal.execute(capability)$$

When this statement is executed is Girji, Girji will first check if the capability is tampered with. Girji does this check by comparing the signature of the capability with the value which is calculated recursively using HMAC [21]. If the signature is checked correct, Girji will start executing sequentially the code in each item of the policy chain. Prior to that, Girji will first evaluate the restriction to see if the operation is allowed. If the constraints are evaluated to hold true, the code is executed. Then Girji repeats the constraints evaluation and code execution for the second policy item, the third one, until the last item in the chain. Since the code generated by the principal might be malicious, Girji's capability execution runtime has to make sure the code is executed in a secure manner. When executing the code of each operation of the policy item, a profile is applied to each code execution runtime. The profile lists what the code can do. For example, a network connection code is executed failed if a profile, which does not state a network connection is allowed, is applied to the code execution runtime. Besides that, the code must follow a input/output invention that is the code can read only from the input file and should write output only to the output file. The reason for that is that each code execution should read only the input yielded from the code of the previous policy item in the chain. This mechanism also restricts the code from reading someone else's data.

### 4.3.1 Policy Chain

A capability is executable in Girji in that the *policy chain* contains a number of links to the source code of operations. A key difference between code consent object and capability is that there is a *policy chain* in the capability. A *policy chain* is a list of policies which are added when constructing or delegating. Since all the operations are contained in the policy chain and the capability is stored in principal's side, Girji need not store or manage capabilities. There are some requirements for the policy chain. The constraints should be attenuated from the root item to the end item because the chain represents more restricted access control. Moreover, the operations in the chain are executed in a sandbox so that the presenter can access only the result of the execution of the whole policy chain. The presenter cannot access the results of the execution of the policy items. The reason is for security because if the presenter is able to access the intermediate results, he is able to access more data than he is allowed to, which may lead to information leakage. When delegating capabilities, one or more policy items are added to the policy chain to give

grantee confined access control to the processed result which yielded from the added operations. For the added operations, the input data is the result of the whole policy chain's execution of the old capability. Another function of policy chain is to computer the *signature* of the capability be ensure that the capability is not tampered with during transferring.

A *signature* is a chained of keyed cryptographic digests derived from the policies. The signature is computed using HMAC functions [21] in a nested fashion. A signature of a capability is yielded by the following steps:

- Take the first policy item in the policy list.
- The strings of *codeRef*, *constrains* are concatenated as a long string.
- The long string is truncated to a fixed length (e.g., 128bits or 256 bits).
- The first HMAC key value is computed by the function HMAC(*secret key*, longStrTrunc).

The key is used as the secret key for the second policy item in the policy list and the second HMAC key is computed as the same way as the first key. By this nested HMAC function that is applied from the first policy item to the last one in the policy chain, a chain of HMAC key is also constructed. However, the final HMAC key is set to be the signature of the capability. This mechanism is efficient when a principal wants to delegate a capability. The principal adds a policy item and uses the signature of the capability as the key for the HMAC function, thus the signature of the derived capability is computed by taking the previous capability's signature and the string of policy into HMAC function. In short, the signature of a capability can be used to ensure that the capability is tamperproof and used as the key for construction of the delegated capability.

## 4.4 Reference Monitor

Prior to executing the code in the capability, a *reference monitor* must be invoked to mediate all the operations. A *reference monitor* is a component that verifies every capability and enforces the security policy for every access. The reference monitor concept was introduced to determine what kind of access is authorized [22]. The reference monitor will perform the signature checking and constraints evaluation to prevent unauthorized access. Every access to the data is completely mediated by the reference monitor. The overview of the reference monitor design is shown in Figure 4.4. When a principal presents a

capability, the reference monitor will verify the signature of the capability by comparing it with the HMAC value computed by the reference monitor. If these two values are identical, this means that the capability has not been tampered with. Second, based on the `asId` field, the reference monitor searches for the code consent object in the authorization database and gets the value of `revoked`. If it is true, which means that all the capabilities associated with this code consent object have been revoked, the access is denied by the reference monitor. Otherwise the reference checks the freshness of the constrains and evaluates the constrains in each policy item of the policy chain. Meanwhile, all the events are recorded as logs which are sent to audit trail. If all the constrains of one policy item hold true, the reference monitor will grant the operation in that policy. The same procedure is applied to the next policy item in order. If any policy fails to be verified, the reference monitor will deny the operation. In short, the reference monitor will do the following checks sequentially: 1), verifies the signature; 2), checks if capability is revoked; 3), checks if constrains are updated; 4), evaluates the constrains.

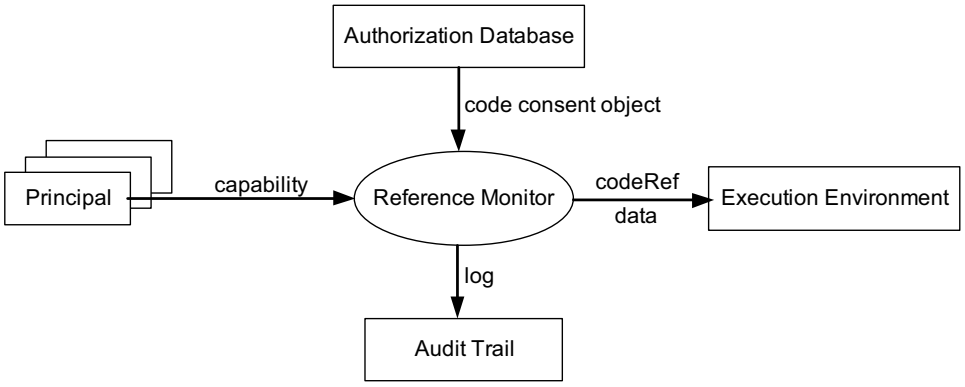



**Figure 4.4:** Reference Monitor

### 4.4.1 Capability Execution Environment

As stated earlier, ASs are not fully trusted by Girji and users. When registering with Girji, it is likely that the operation function, which the AS provides, contains malicious code, e.g., some code tries to access other users' data file or create a socket. Apart from that, an operation may try to consume all memory to break down the whole execution environment. This brute operation could also as an attack. In order to secure the execution runtime, each operation should be executed in a secure and restricted *capability execution environment*, which is essentially a sandbox. In the sandbox, each function can not do any operations on any files other than what it is allowed to. In addition, each operation function should not consume all the computation resources of the execution environment. Besides the constrains evaluated in the reference

monitor, when executing the operation function, a *profile* is applied to execute the untrusted code. A *profile* is a file that consists of a set of rules specified using `AppArmor` syntax. These policies will be enforced on the process in which the operation is executed. We can also specify how much memory and the number of CPU cycles the operation can use. Different profiles can be applied on different operations. With the profile, the operations in the policy chain will be executed sequentially and the latter function can read only the output of the preceding function as its input. As shown below, the example of the *profile* specifies that network connection is disallowed, the process is only allowed to read and write the directory `/tmp/**`, and the process can only use virtual memory up to 1 GB.

```
deny network,   #disallow all networking
/tmp/** rw,
RLIMIT_AS = 1024*1024*1024
```

## 4.5 Capability Revocation

A scalable revocation service [23] is employed to disseminate the list of *capId*s of the revoked capabilities among capability components. After bootstrap, a mesh is generated among all the capability components by a hash function. As is shown in Figure 4.5, each capability component has the same role and acts as a back-to-back agent which receives message from other components while sending messages to its neighbors. Therefore, each capability component is connected to more than one components, but not fully connected as fully connection will not scale.

When a user revokes a capability, a list which includes *capId* of the revoked capability, is created by the capability component. This component pushes the *capId* list to a set of capability components running *fireflies* [24] [25] agent. Then the list is eventually distributed to all the components on the overlay network. Thus, by gossiping all the components have a list of *capId*s of all the revoked capabilities. When receiving new message, if the version number is greater, which means the list is more up-to-date, the current list is replaced by the received list. The sending thread periodically picks one capability component from its view. If the version number of the current list has not been sent to the component by comparing the version number with the one which is sent last time, then the current list is sent to the component.

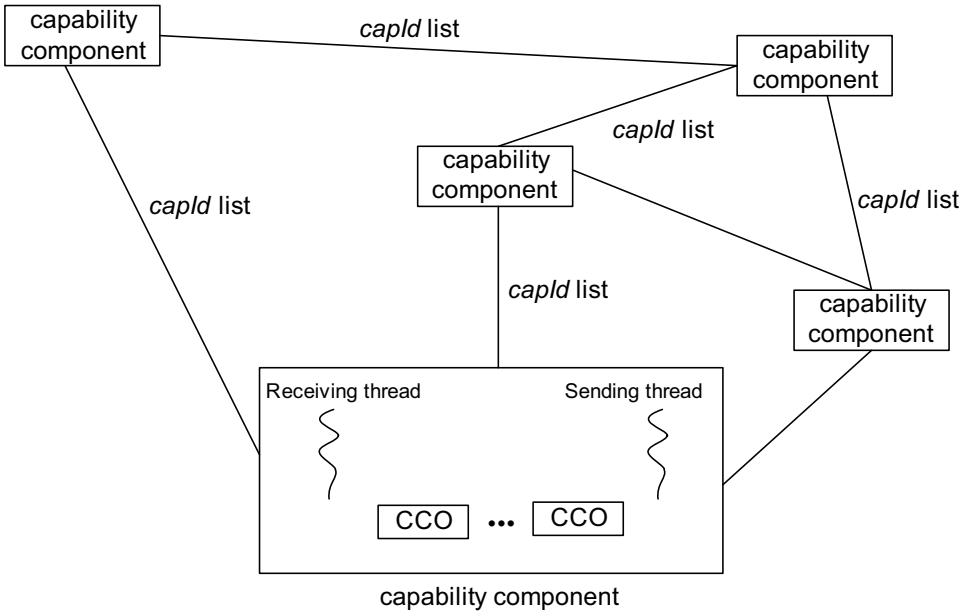



**Figure 4.5:** The overlay network of capability components

## 4.6 Implementation Details

In our prototype implementation, Girji is able to acquire the body-sensor data from RunKeeper web service via OAuth 2.0 protocol. Different types of data are stored in individual Comma Separated Value (CSV) files. For instance, all the data of fitness activities is stored in `fitness.csv` file. All the Comma Separated Value (CSV) files associated with a user are stored in the same directory which is the user's *infospace*. We have also developed a portal for users to create, delegate a capability to a user who is even in a different administrative domain. In addition to that, we have implemented a simple reference monitor to verify capabilities. With respect to the execution of capabilities, all the source code of operations is written in R language. Therefore, we use OpenCPU [26] as a execution environment that executes R package by receiving HTTP request. We have developed several R packages for the common operations such as filtering out all the `running` data of a user, selecting only the data of the specified columns, aggregating and so on.

### 4.6.1 Capability Construction

Both users and ASs can construct a capability. When requesting data access, the AS could set `codeRef` in the informed consent by selecting the operations provided by Girji. For example, Girji develops R packages for common operations e.g., retrieving only the specified activity data from the user's *infospace*,

and retrieving the activity data of the specified year or month, etc. If an AS wants to acquire the user's running data in 2014. The AS could specify the operation by combining the two operations provided by Girji. The value of `codeRef` of the informed consent would be a list of links to the corresponding R packages:

```
/ocpu/library/girji/activity
/ocpu/library/girji/time
```

After the user specifies the `constraints` and signs the consent, a code consent object and the corresponding capability is constructed by the following steps:

1. The constructed code consent object is shown below.

```
   String CCOId = 69i57j0l52782j0j7;
String provider = testAS;
String consenter = wei@uit.no;
boolean revoked = false;
List operations =
      { {"/ocpu/library/girji/activity", "Running"},
        {"/ocpu/library/girji/time", "2014"}
      };
           int constraints.version = 0;
               int constraints.ttl = 12;
   String constraints.accessPeriod = "8:00-12:00";
String constraints.allowDelegation = true;
```

2. Girji initiates a new capability by setting the fields of the capability. `capId`, `asId` are unique in Girji and `userId` is the user's account name in Girji.

```
capId=get_wei_running_2014;
description="Retrieve wei's running data in year 2014";
asId=testAS;
userId=wei@uit.no;
```

3. Since the AS specifies two operations in the informed consent, the policy list has two policy items. Girji populates the policy list by setting the operations and constrains as follows.

```
policy[0]
    codeRef="/ocpu/library/girji/activity";
    param="Running";
```

```
        constraints.ttl = 12;
        constraints.accessPeriod="8:00-12:00";
        constraints.allowDelegation=true;
        constraints.version=0;
    policy[1]
        codeRef="/ocpu/library/girji/time";
        param="2014";
        constraints.ttl = 12;
        constraints.accessPeriod="8:00-12:00";
        constraints.allowDelegation=true;
        constraints.version=0;
```

4. In order to ensure the policies are not tampered by eavesdroppers, a signature is computed using HMAC functions.

   key = HMAC( key, $str_{policy}$ )

   concat is the function that concatenates each string of the policy item. The HMAC function is executed two times because there are two policy items in the policy list. First, Girji initiates $str_{policy0}$ with of the value concat(codeRef, ttl, accessPeriod, allowDelegation) Second, with the secret *root key* which is only known to Girji, a key is computed by:

   $key_0$ = HMAC( root key, $str_{policy0}$ )

   The derived key (i.e., $key_0$) is used as the HMAC key for the next policy item. Thus, $key_1$ is computed nested using HMAC and is set to be the signature of the capability since $key_1$ is the final key.

   $key_1$ = HMAC( $key_0$, $str_{policy1}$ )

5. A capability is created with the file name get_wei_running_2014.xml. The file's content is readable and is shown in Figure 4.6. The capability will be sent to the recipient through email.

Girji provides an HTTP front-end for principals to construct a new capability or delegate any existing capability. The user has to specify not only constraints, but also the codeRef field. Girji provides source code of some shared common operations, such as the operation of retrieving a user's positional data during specified time period, and the operation of aggregation of a type of data. With these operations which come with the Girji service, one is able to construct a capability simply by clicking few buttons in the Girji portal. However, if one wants to have customized operation, he needs to write the source code by

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<capability>
    <capId>get_wei_running_2014</capId>
    <description>Retrieve wei's running data in year 2014</description>
    <asId>testAS</asId>
    <userId>wei@uit.no</userId>
    <policies>
        <policy>
            <operation>
                <codeRef>/ocpu/library/girji/activity</codeRef>
                <param>running</param>
            </operation>
            <constraints>
                <accessPeriod>8:00-12:00</accessPeriod>
                <allowDelegation>true</allowDelegation>
            </constraints>
        </policy>
        <policy>
            <operation>
                <codeRef>/ocpu/library/girji/time</codeRef>
                <param>2014</param>
            </operation>
            <constraints>
                <accessPeriod>8:00-12:00</accessPeriod>
                <allowDelegation>true</allowDelegation>
            </constraints>
        </policy>
    </policies>
    <signature>f01dc006bc1f7db2d4d7b46fcfb4e8c71a2c4bc5</signature>
</capability>
```

**Figure 4.6:** An example of a capability file

himself and upload it onto Girji so that he is able to get the `codeRef` of this operation.

### 4.6.2 Capability Delegation

A new capability is derived when a user or an AS delegates an existing capability to others. Girji provides an HTTP front-end for facilitating capability delegation. While capability delegation is very much analogous to capability construction, one needs to add policy items to the policy list and compute a new signature for the derived capability. Further, one should assign a new `capId` to the capability. For example, if a user wants another researcher to see his monthly aggregated running distance in 2014 in a pdf file, the user could add one policy item to the policy list by selecting the operation and set access constraints of the added policy item. After that, the user delegates the derived capability to the researcher. The user would do as follows:

1. The user goes to `Delegate a capability` page in Girji, and assigns a new unique `capId` to the new capability. The user can not modify the `asId` and `dataOwner` fields because the original capability is associated with the code consent object constructed by the AS and the data owner.

2. The user adds one more policy item to the policy list of the original capability. The access period of this operations is attenuated to `10:00-11:00` and the derived capability can not be delegated further.

```
codeRef="/ocpu/library/girji/monthly_aggregated_distance_p
        df";
param="";
constraints.accessPeriod="10:00-11:00";
constraints.allowDelegation=false;
```

3. A new signature is computed using HMAC by taking the signature of the original capability as a key, and the string concatenation of the new policy item's fields.

$$signature_{delegated} = \texttt{HMAC}(\ signature_{origin}, \ str_{policynew}\ )$$

4. `monthly_aggregated_distance_pdf.xml` is created and sent to the researcher through email.

It is common that a principal just wants others to see the aggregate values of the data. Or if another researcher wants to access the result of the principal's operation, the principal could delegate the capability to the researcher. The capability delegation is done by creating a new capability, then setting the operation and restriction and adding the policy item to the end of the policy chain. After calculating the new signature of the policy chain, a new capability is derived so that another principal can execute the policy chain in the capability to access the desired result. One thing to mention, since the restriction is attenuated from the root policy item to the end item, the new added restriction should not be looser than the preceding restrictions. Besides that, if the principal thinks the operation is confidential, it could set the value of the operation to be the internal link to the code which others are not able to see it. The most important thing for the other principal is the output of the operation, not the code of the operation. Let us following the above example, by executing the capability, $P_a$ is able to access `tom`'s running data in `2014`. If another principal $P_b$ would like to access `tom`'s aggregate running distance in each month, $P_a$ will create a new capability, write code to output the aggregate distance and set the *codeRef* field to be the link address of the code. Since $P_a$ can only access the running data during `8:00-17:00`, the restriction for the new capability should be either the same as the previous restriction or

tighter that that, for instance `9:00-12:00`. After that, $P_a$ needs to calculate the new signature by

`HMAC(capability.signature, strconcat(`$restrcition_{new}$`,` $operation_{new}$`);`

With new signature and new capability chain, the process of capability delegation is finished. Therefore, $P_b$ is able to access `tom`'s aggregate monthly running distance data in 2014.

### 4.6.3 Capability Verification

When a principal presents a capability, the reference monitor will perform verification on the capability. The verification includes 1)signature verification; 2)revocation check; 3)constraints freshness check; 4)constraints evaluation. For example, after the capability `get_wei_running_2014.xml` was created, the user `wei@uit.no` modified `accessPeriod` in the code consent object to `9:00-11:00` so that `constraints.version` was incremented to `1`. When the AS presents the `get_wei_running_2014.xml` to the reference monitor, the following verifications are performed.

1. Girji will compute the signature by taking the *secret key* as the key of HMAC function. After a chain of nested HMAC calculation, a final key is yielded. Girji compares this final key with the signature of the capability (i.e., `f01d**4bc5`). If two values are identical, this means the capability has not been tampered with. Otherwise, the AS will be prompted with a `signature verification error` message.

2. Girji checks the value of `revoked` in the code consent object as shown in Subsection 4.6.1. If the value is `false`, the capability is still valid. Otherwise, a message of `capability not valid` will be shown to the AS.

3. Girji compares `constraints.version` with the one in the capability. Since the user modified the constraints, the value of `constraints.version` is greater than the value of the one in the capability. Girji will update `accessPeriod` from `8:00-12:00` to `9:00-11:00`.

4. Girji evaluates all the predicates of each policy item. If the results hold true, the `codeRef` and `param` will be executed in the OpenCPU sandbox. The combination of predicates evaluation and execution are performed from the root policy item to the last item.

Prior to the execution of the capability, Girji will validate the capability by checking if the signature in the capability is identical with one calculated using its `secret`. If the values are identical, this means the capability is not tampered with during transferring. Otherwise Girji will raise *validation exception* and stop execution. The reason for validation is that the principal who holds the capability can modify any field of the capability because the capability is essentially a collection of string. What's worse, the principal can modify the operation to another operation including malicious code. In order to yield the correct output, the capability validation is performed prior to execution.

### 4.6.4 Capability Execution Model

Whenever the principal would like to access the user's data, the principal just executes the capability in Girji. What the principal has to present is only a capability. The primary data-access API call provided by Girji has the form:

$$principal.execute(capability)$$

In order to get the output of the execution of the whole policy chain, The principal must present the capability with the correct signature to the reference monitor in Girji. After verifying the capability's signature, the reference monitor checks if the capability has been revoked and if the constraints have been modified by the data owner. The capability verification module is tightly coupled with the capability execution module, as prior to the execution of the operation, the reference monitor will evaluate the constraints against the context.

After the reference monitor performs the capability verification, the capability is executed in the OpenCPU server, which is essentially a sandbox. For the operation in each policy item, the reference monitor will send an HTTP POST request with the parameters and data file attached in the request body. OpenCPU will create a separate process to execute the code which is specified in *codeRef*. Then it will respond with `201 Created` message on successful POST request. Meanwhile, an execution session is created so that the reference monitor is able to retrieve any object in the session such as intermediate datasets, R objects, or output result in different formats (e.g., json, text, html, pdf, etc). The information of the session can be retrieved by HTTP GET request. The information of process session 1 in Figure 4.7 is shown as below:

```
/ocpu/tmp/x0b26c3cf/R/.val,
/ocpu/tmp/x0b26c3cf/source,
/ocpu/tmp/x0b26c3cf/console,
```

```
/ocpu/tmp/x0b26c3cf/info,
/ocpu/tmp/x0b26c3cf/files/output0.csv,
/ocpu/tmp/x0b26c3cf/files/fitness.csv
```

The output file yielded from the execution is `output0.csv`. Thus, the reference monitor retrieves this information by sending a HTTP GET request to the url of the session. The object will be sent in a `200 OK` response message if the GET request is sent successfully. After receiving the `output0.csv`, which is in turn the input file when executing *codeRef* of the second policy item. The information of process session 2 is shown as below:

```
/ocpu/tmp/x0b1438cf/R/.val,
/ocpu/tmp/x0b1438cf/source,
/ocpu/tmp/x0b1438cf/console,
/ocpu/tmp/x0b1438cf/info,
/ocpu/tmp/x0b1438cf/files/output0.csv,
/ocpu/tmp/x0b1438cf/files/output1.csv
```

The intermediate output files are not accessible to the principal unless they are the output files of the execution of the last policy item. The result output file of the capability executed in Figure 4.7 is `monthly_distance.pdf` as it is the result of the last policy item. The information of session 3 is shown as below:

```
/ocpu/tmp/x0b1420dd/R/.val,
/ocpu/tmp/x0b1420dd/source,
/ocpu/tmp/x0b1420dd/console,
/ocpu/tmp/x0b1420dd/info,
/ocpu/tmp/x0b1420dd/files/output1.csv,
/ocpu/tmp/x0b1420dd/files/monthly_distance.pdf
```

When executing the `codeRef` of each policy item, the OpenCPU server will create a separate process for each policy item. In order to ensure that the code of each policy item is executed in a limited environment, a *profile* is applied on the process. The *profile* defines what system calls can be performed, what directories can be accessed, how much memory is available for the runtime and so on. With this profile, when executing the code trying to access the data files of any other user, Girji will raise a *profile not permitted* exception because the profile did not include any other user's directory. After executing the policy chain, the output of the last policy item is the data the principal can access. The *profile* specifies what operations can be performed on which files. In addition, each function can read only the output of the preceding function as its input. With the combination of the *profile* and the rule, the function in each policy item is executed in a secure and restricted environment. If a principle

wants to execute the capability `monthly_aggregated_distance_pdf.xml`, the flow of the capability's execution is shown in Figure 4.7.

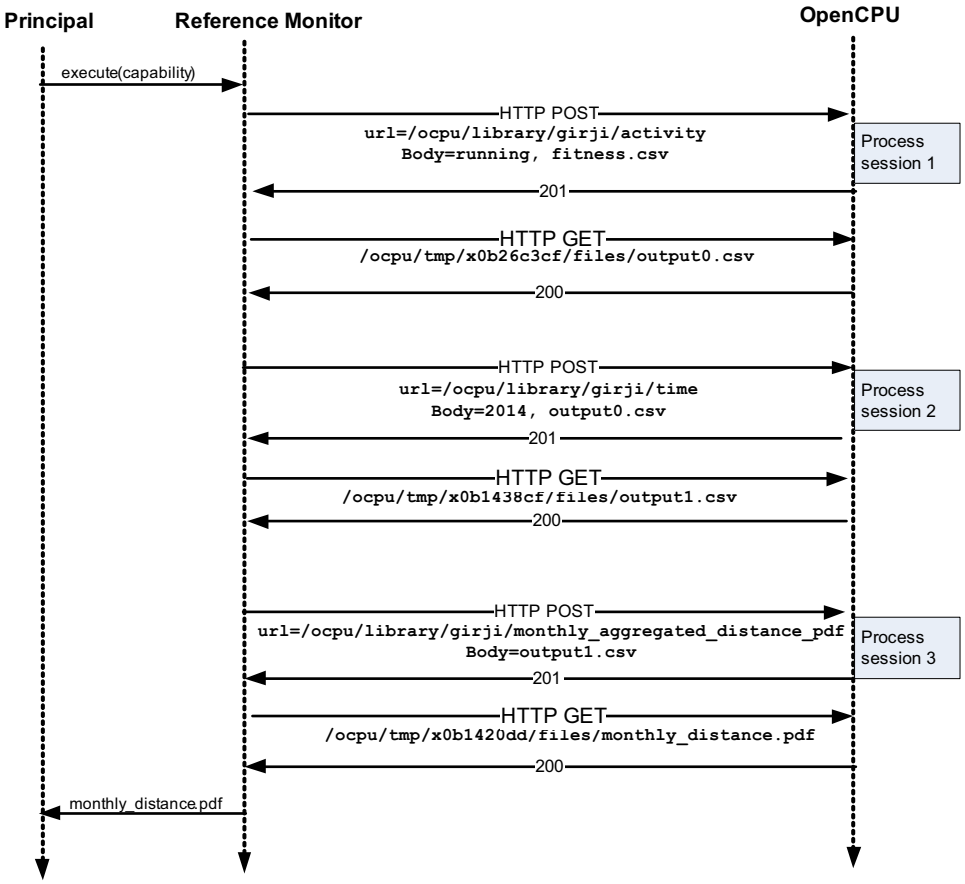



**Figure 4.7:** Execution flow

In the above example, the data that the principal can access is `tom`'s running data in 2014. The principal can do whatever operations, but only on the specified data, because the output of the capability execution is the only input of the principal's operation.

In order to demonstrate the reference monitor, we wrote each function in `girji.R`, then built it and deployed the package on OpenCPU server. The name of the package is `girji`. Figure 4.8 shows an example of the execution chain of the capability `monthly_aggregated_distance_pdf.xml`, which has three policy items in the policy chain. After the signature of the capability is verified and the list of constraints are evaluated to be true, the reference monitor will send the input file (i.e., `fitnessActivity.csv`) and the parameter (i.e., `'Running'`) to the OpenCPU server and execute the code specified in the *codeRef* by sending an HTTP POST. When executing the `R`

source code in OpenCPU server, a security profile is applied onto the execution so that malicious or resource-intensive computation is denied by the execution. After executing `activity` function, the output file `activity.csv` becomes the input file for the next function (i.e., `time`) which is specified in the *codeRef* of the second policy item. The results of the intermediate functions are not accessible to the principal. After executing the function `monthly_aggregated_distance_pdf`, a pdf file `monthly_distance.pdf` is created. Since this function is the operation of the last policy item, as the result of the execution of the capability, `monthly_distance.pdf` is accessible to the principal.

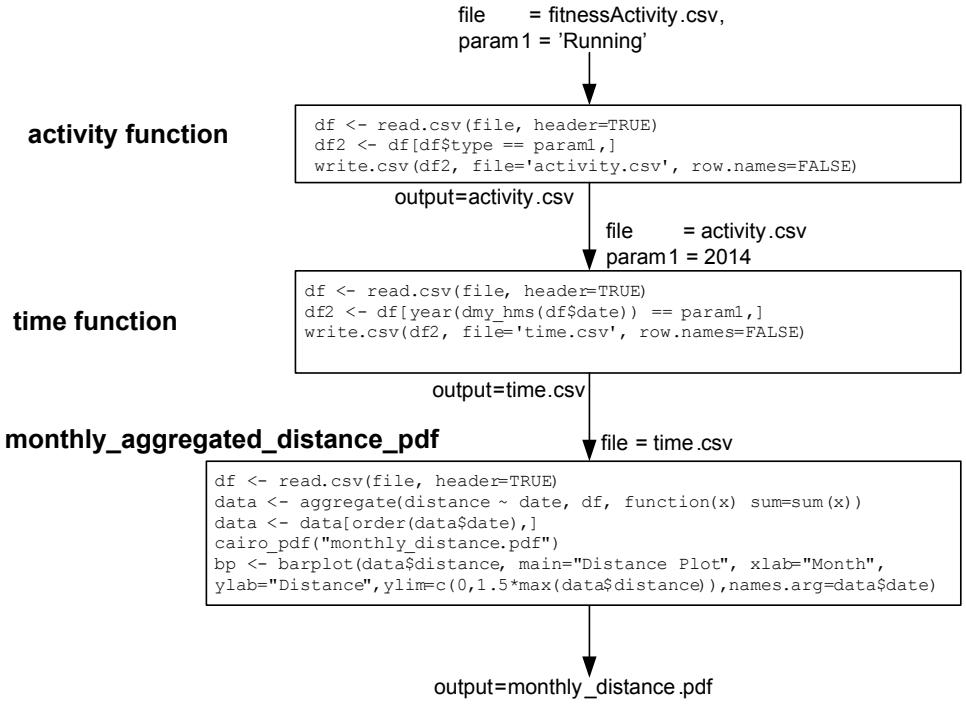



**Figure 4.8:** Execution chain

As for the grantor, he shares his aggregated running data with the grantee by adding a function of aggregating the distance for each month and delegating the capability to the grantee. Instead of sharing the all the columns of raw data (i.e., time, distance, calories, duration), the grantor shares the converted data so that less information (i.e., sum of distance) is disclosed. In other words, the grantor shares with the grantee only the operation of aggregating his running distance for each month. This confined sharing ensures the security of access control to his data. Therefore, the grantor achieves the aim to share his data in a confined manner. As for the grantee, it becomes much easier to get access to the grantor's aggregated distance as the grantee possesses the capability which entitles him to access the data. Even the grantee is outside

Girji, he is still able to execute the capability. Since the raw data may change as the grantor finishes more activities, the grantee can still get the updated pdf graph by executing the same capability. Therefore, the grantee is able to access the dynamic result of the capability's execution with no need of updating capability.

# 5 Evaluation

This section details a use case for measuring the end-to-end latency during the time between the user executes a capability and the user gets the output in Girji. We also discuss the results of the measurements and identify the bottleneck of a capability's execution.

## 5.1 Case Study

We now examine a specific use case which enables an athlete through capability delegation to share his dynamic aggregated running distance with other persons. The athlete $A$'s running activity is captured by the RunKeeper app and the distance data is stored in RunKeeper web service. In order for Girji to retrieve the athlete's runkeeper data, $A$ needs to login to Girji and connect Girji application to $A$'s RunKeeper account via OAuth protocol. After Girji has access to $A$'s account, his fitness activities in RunKeeper will be pulled to and stored in his infospace in Girji. Therefore, $A$ is able to give analytical services access to his data so that analytics can be performed on the data. Since the athlete $A$ wants to get his own information of running distance, $A$ needs to register an analytical service. In this case, $A$ is not only the owner of the data but also the provider of the analytical service. $A$ could create a self-signed code consent object by choosing system-provided operations because Girji already provides *codeRef*s of several off-the-shelf operations, such as filtering out a specific activity from all fitness activities, retrieving the data in the spec-

ified year, etc. There are no `constraints` in the code consent object as $A$ is the owner of the data. A code consent object is constructed after the athlete $A$ signs the consent. Subsequently a capability $cap_a$ is also constructed from the code consent object. There are two operations, which is shown in Table 5.1, in the code consent object as well as in $cap_a$.

| **CodeRef** | **Param** |
|---|---|
| /ocpu/library/girji/activity | Running |
| /ocpu/library/girji/time | 2014 |

**Table 5.1:** Operations in the code consent object

By executing $cap_a$, $A$ is able to get the detailed information (e.g., specific time, distance, duration, calories) of his running activity in 2014. As the data owner, it is fine for $A$ to see all the details of the activity. However, if $A$ wants to share the running information with another person $B$, who might even not be a user in Girji, $A$'s raw data may be leaked out by $B$. To reduce the risk of disclosing raw sensor data which might contain sensitive information, instead of sharing raw data, $A$ can share only the aggregated data by delegating $cap_a$ to $B$. Since the output of the execution of $cap_a$ is $A$'s running information in 2014, $A$ could add one more policy item to the policy chain to aggregate the running distance and generate only a graph. $A$ could also specify the constraints in the policy item to further restrict the availability of the operation. With the new capability $cap_b$ which was delegated from $cap_a$, $B$ is able to get a graph by executing the $cap_b$ in Girji. Even if the user $B$ is not registered in Girji, which means $B$ is from different administrative domain and $B$ can not login to Girji, $B$ is still able to execute $cap_b$ on the front page of Girji. The output of $cap_a$ and $cap_b$ is shown in Figure 5.1. Through $cap_b$, $B$ is able to access $A$'s only aggregated data, not raw data or $A$'s other data. After Girji acquires $A$'s updated `Running` activities, $B$ is able to get the dynamic graph by simply executing $cap_b$ again.

This use case demonstrates that our prototype implementation complies to the principle of least privilege. For the user, he can access his own data only through a capability. If he wants to share some of his data with others who may be even from a different administrative domain, he can simply delegate existing capability with additional operations and restrictions, or create a completely new capability. The operations in the policy chain of the capability are what the grantee can do. Moreover, with the same capability, the grantee can get the dynamic result of executing the capability in that the operation is still the same, only the raw data changes.

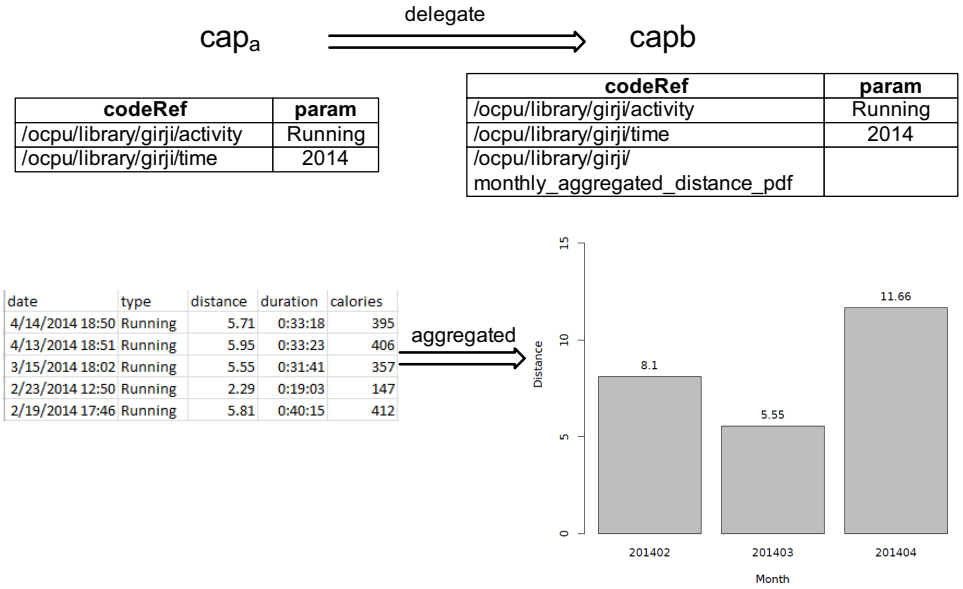



**Figure 5.1:** Output of capabilities

## 5.2 Experiments

In order to evaluate the performance of our prototype, we performed several experiments to examine if our implementation can work in the real world. We tested the data transfer performance of RunKeeper web service to check if it is capable of handling Girji's upload and download requests with large number of data points. Experiments are also conducted to evaluate the end-to-end latency of executing a capability. We measured the verification time, code execution time as well as result download time. By analyzing these three types of latency, which constitutes the end-to-end latency, we identified the bottleneck of a capability's execution.

### 5.2.1 Experiments setup

The prototype of Girji was tested in a cluster of computers which are in the same network. Girji consists of two parts: one part, which is the R sandbox (i.e., OpenCPU), is installed on a Ubuntu 12.04 Server with Intel Core 2 3.00GHz with two cores, and with 4 GB of RAM; the other part, which is the back-end server, is installed on a 2.4GHz, four core (Intel Core2 Q6600 processor) system with 4 GB of RAM, running Windows 8. All the test cases are executed ten times to get more objective results.

### 5.2.2 Data Transfer Time of RunKeeper

It is common that an athlete runs with the RunKeeper app a long distance which will result in many sequence geographical points along the route. The reason for doing this experiment is that we would like to see how much time it takes for Girji to acquire the data from RunKeeper web service after the user finishes an activity. We evaluated how long it becomes available to users after Girji uploads an activity which was captured by other sensors for instance ZXY. With the HealthGraph API, we were able to generate JSON files which have the same structure as that is shown below:

```
{
   "type": "Running",
   "equipment": "None",
   "start_time": "Sat, 1 Jan 2011 00:00:00",
   "notes": "My first late-night run",
   "path": [
   {
      "timestamp": 0,
      "altitude": 0,
      "longitude": -70.95182336425782,
      "latitude": 42.312620297384676,
      "type": "start"
   },
   {
      "timestamp": 8,
      "altitude": 0,
      "longitude": -70.95255292510987,
      "latitude": 42.31230294498018,
      "type": "end"
   }
   ],
   "post_to_facebook": false,
   "post_to_twitter": false
}
```

The `path` field of the structure is a list of data points captured by RunKeeper app. We uploaded via HTTP POST the files with the number of data points from 5000 to 30000 and downloaded via HTTP GET the files again. We restarted the whole system and remove all the intermediate files after each test run. As can be seen in Figure 5.2, while the download time does not increase much, the latency of uploading time is significant when the number of data points increases. If the sampling rate of RunKeeper app is 10 Hz, a soccer player will generate 54000 data points in a soccer game, the upload time of this number

of data points will be much more than one minute. Further, the precision rate of consumer-level sensors is much lower than professional ones like ZXY that captures positional data at a rate of 20 Hz in real time for all 22 players in a soccer match. With ZXY a player will generate 108000 data points which are two times more than the number of data points generated by RunKeeper app. If Girji uploads the activity of a soccer play, it may take hours to finish it.

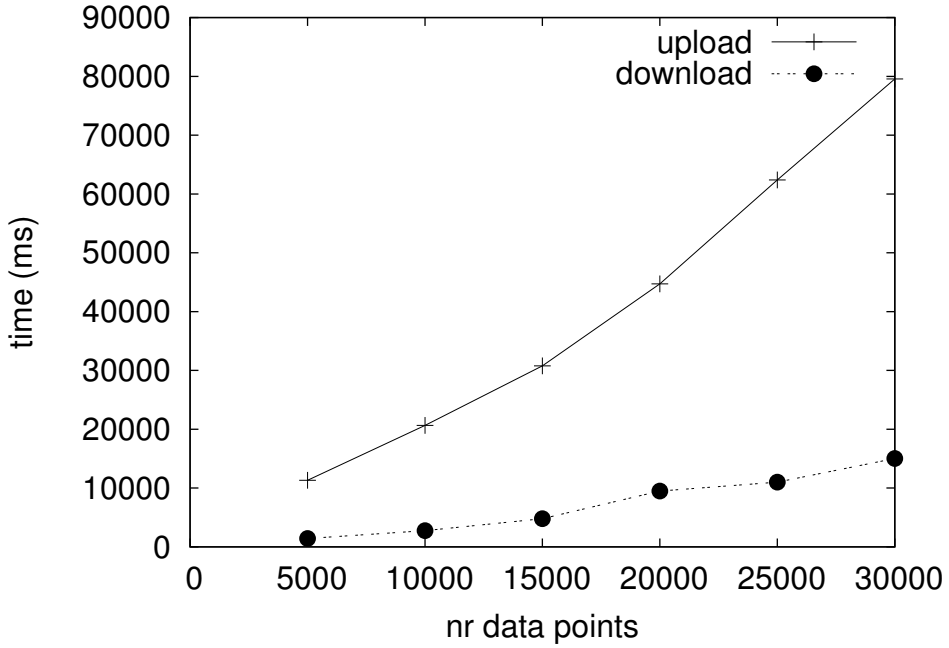



**Figure 5.2:** Data transfer capacity of RunKeeper

Although the uploading incurs high latency, the download latency is relatively low. As is shown in Figure 5.2, the time of downloading a record with 30000 points is roughly 14 seconds. Girji could have a background-running acquisition agent periodically pull the data from RunKeeper. Every time the agent retrieves data from the user's RunKeeper account, only the updated data is pulled.

### 5.2.3 Capability Execution Time

The end-to-end latency of executing a capability is the delay from the time when the user clicks `execute` button to the time when the user sees the result. When executing a capability, Girji's back-end will first verify the capability's signature, then execute the operation in each policy item, finally download the result file from OpenCPU so that the user can see the result. The last two

steps are performed repeatedly for each policy item in the policy chain of a capability. Therefore, we would like to examine if the end-to-end latency is affected by the number of policy items. We generated three capabilities (i.e., $cap_{one}$, $cap_{two}$, $cap_b$ ) which have the same functionalities but are split into different number of parts. Besides, the execution of a capability incurs network communicate as the back-end and OpenCPU are installed on separate machines. We generated several data files with different number of data records on which the operations will be performed. $cap_{one}$ has only one policy item in the chain because the customized code is written by the user himself. $cap_{two}$ has two policy items whose *codeRef*s are both provided by Girji. After executing these three capabilities, the same pdf graph will be received. Figure 5.3 shows the end-to-end latency of executing capabilities with different number of data records and policy items.

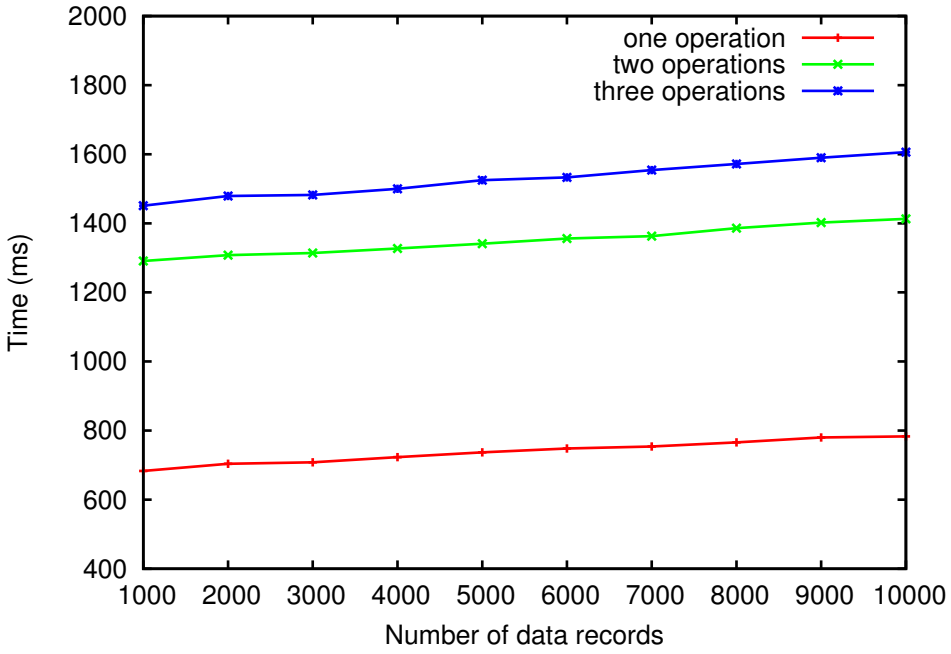



**Figure 5.3:** Execution time of a capability

From Figure 5.3, we found that the end-to-end latency increases linearly as the number of data records increased. For a given capability, the input data files with different number of records did not affect the latency much and the latency was stable. However, the overall latency increased as the number of policy items increased as more operations were executed. While the capability with fewer policy items reduces the end-to-end latency, the analytical service providers have to write more customized code. The ideal situation is that every capability has only one policy item in the chain. Then all analytical

service providers have to write the source code by themselves. But still, after delegating, a capability also has more than one policy item.

### 5.2.4 Minimum Overhead for Each Policy Item

In order to examine the scalability of the implementation, we executed a test case which measured for each policy item the verification time, code execution time as well as download time. We generated a dumb function which essentially did nothing but returned 0.

```
function ()
{
    0
}
```

There was no input data file to be uploaded and passed to this function. The reference of this function was set to be *codeRef* of each policy item so that every capability with different number of policy items all had the same operation. Two hundred capabilities were constructed with the number of policy items from 1 to 200. The test was repeated ten times. In this test case, we did not restart the service after each test run because OpenCPU supported cache and we would like to evaluate the overhead when the system was warm. In Figure 5.4, we found that the following interest results. First, the verification time for each policy item was about 57 us and the time did not change as the number of policy items changed. Second, the time of executing code and download time were stable. At the beginning there was some jitter in the code execution time and download time as there might be some scheduling in OpenCPU. The execution and download involved network communication but the signature verification did not. Another reason is that when the capability was executed for the first time, the dumb function was not cached in OpenCPU. Thus, it took more than one hundred milliseconds to execute the function and about ten milliseconds to download the result file. After the function was cached, the system became warm and the execution time dropped by an order of one hundred times. We also observed that the latency of downloading is higher than that of executing in that the function was cached but the result was not. Every time a function was executed, a new session was created and the result was written into a file. When Girji downloaded the result, OpenCPU had to load the file and sent it to Girji's back-end.

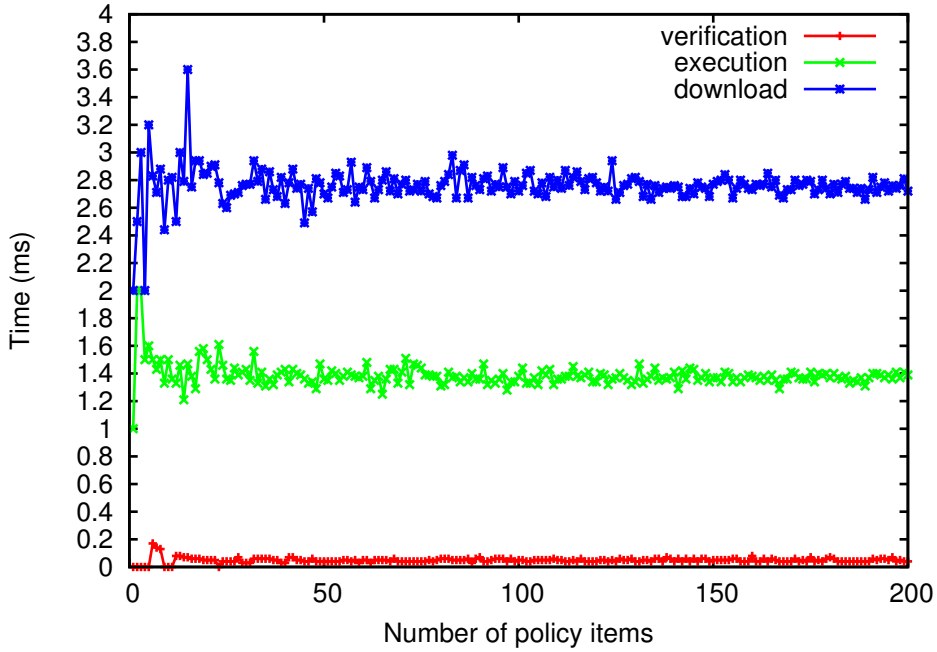



**Figure 5.4:** Minimum overhead for each item

### 5.2.5 End-to-end Latency Analysis

To take a deeper look at the end-to-end latency, we conducted another experiment which measured respectively the execution time of a policy item and download time of a output file as the number of data records increases. As can be seen in Figure 5.5, we observed that the time of executing a policy item is significantly more than the download time. Therefore, it is identified that the bottleneck of the end-to-end latency is the time of executing each policy item. Meanwhile, the number of data records does not affect the end-to-end latency much. The average latency of the capability with three policy items is 200 milliseconds more than the average of the capability with two policy items.

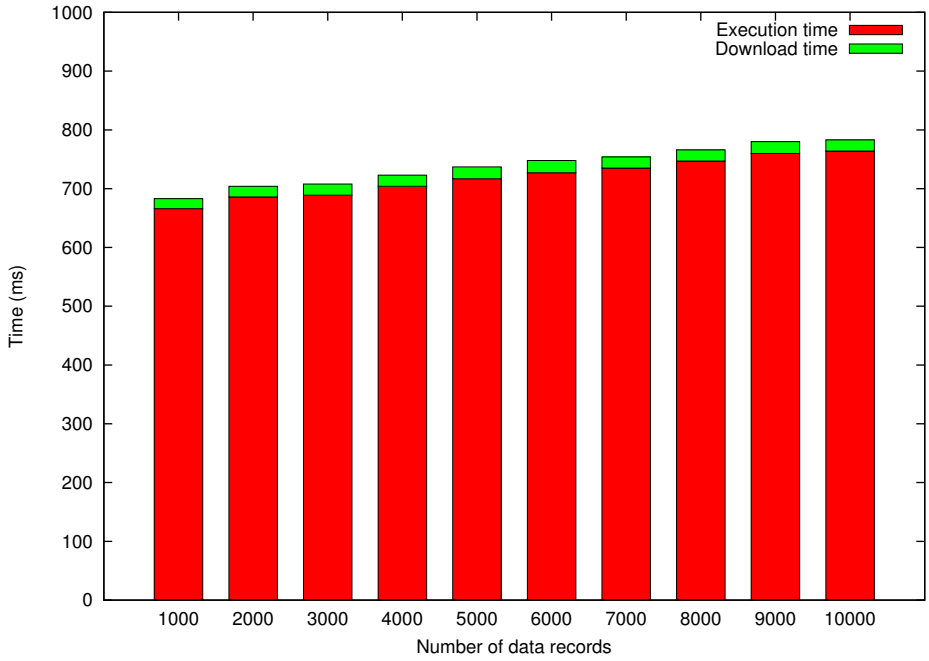



**Figure 5.5:** Latency analysis

# 6

# Conclusions

In this thesis, we have introduced Girji, an infrastructure that is able to converge user's body-sensor data from different sources and securely store the data in long term. In addition, the data of different users is stored isolated in each user's *infospace*. By providing the computation environment in Girji, analytical services (i.e., ASs) are able to apply analytics on users' data from a wide range of body sensors by writing R source code and uploading the package to Girji. To address the challenge of controlled sharing, we present an approach which combines informed consent and capabilities to provide users with greater and more fine-grained control over their body-sensor information disclosure. With code consent capabilities, users are able to share their data in a more fine-grained way by specifying *dataRange*. Users are also able to confine the operations on their data by specifying *constraints* whose value will be copied from the informed consent to the capability. The policies in the capability are enforced when the capability is executed in the reference monitor. The benefit for ASs to use capabilities is that ASs are able to collaborate a lot more with each other by delegating the capabilities of which the execution results are reusable to other principals. The capabilities can be revoked very easily at any time by gossiping among the capability components the list of *capId*s of the revoked capabilities.

In our prototype implementation, Girji is able to retrieve a user's information in RunKeeper back-end via OAuth protocol. With the capability `wei_running_2014.xml`, a user can easily share his fitness activities in RunKeeper with the AS which in turn filters out the user's running activities in

2014. By capability delegation, instead of sharing his raw data, the user is able to share only the graph of his aggregated running distance in 2014. We also have described a specific use case and executed several experiments to evaluated the performance of our implementation. We have observed that the end-to-end latency of executing a capability increases linearly as the number of data records increase. With the capability delegated in the use case, the time of executing the delegated capability to generate the aggregated data from 10000 data records is only 1.5 second. Thus, the performance is good. By measuring the time of capability verification, code execution, and file downloading respectively, we identify the bottleneck is code execution time as the time of data file uploading takes 10 times more that download time.

## 6.1 Achievements

The paper about our work has been accepted by the conference IEEE ISSNIP 2014. In this paper, we have focused on designing an access control mechanism which makes authorization much easier and more fine-grained. We present code consent capabilities to provide owners of body-sensor data with a user-centric control of their data. With code consent capabilities, which reflect the terms of informed consent, ASs are able to perform the operations in the policy chain to access the result data. Principals (i.e., users, ASs) can delegate the authority without the intervention of administrators. The easy authority delegation relieves their labor a lot particularly in a large system. Moreover, principals can collaborate with others, who are outside of Girji, by delegating capabilities so that the recipient principals are able to reuse the result yielded from the capabilities. We have also addressed the requirement of privilege confinement. Each principal can add constraints, which will be enforced when presented in Girji. With the OpenCPU sandbox, all the operations will be executed in a limited process such that the principal can only access the result data of the execution of the whole capability. Therefore, authorization confinement is fulfilled.

The implementation is secure and feasible. First, a user's body-sensor data is stored in his own *infospace* which is completely isolated from others'. In addition, each policy item in the capability is hashed by HMAC with `secret keys` to make sure no principal can tamper with it to modify operations in the capability. Further, by applying a security *profile* to each process, the proper sandbox OpenCPU executes the code in a restricted manner in terms of system calls, memory use and CPU cycles. Last, the execution is completely transparent to principal so that only the result of whole capability's execution is accessible to the principal. By combining these implementation details, we

are able to make the authorization secure. The experiments show that the end-to-end latency of executing a capability is only at an order of second, so that the implementation is feasible in practice.

## 6.2 Related Work

The term *capability* was first introduced by Dennis and Van Horn in 1966 in [27] and is known as an unforgeable token of authority. A lot of capability-based operating systems have been built such as E system [28], EROS [29], and Amoeba [30]. While there are a lot of capability-based models ([31], [32], [33], [34]) which focus more on the kernel level, our focus in on the application level as same as [35]. The difference is that [35] combines capabilities with ACLs while we develop a pure capabilities system. In a distributed system where there may be different protect domains, Kerberos version 5 [36] addressed this requirements with the help of a trusted third party. By contrast, the same requirement is accomplished in Macaroons [20] by discharging a third-party caveat in a public service rather than a proxy. Personal Data Vaults (PDVs) [37] uses *Granular ACL* and *Trace-audit* to give end-users active and more fine-grained control on their sensitive data. Our infrastructure is built on early work to create a more flexible authorization across distinct administrative domains.

Many of the health care systems (for instance [38]) employ the traditional ACLs approach to manage access control to patients' Electronic Health Records. A more advanced Role Based Access Control mechanism [39] enables ease of management as in health care context there are many roles such as health care takers, nurses, doctors, etc. Although we use informed consent which is usually in health care context, we decide to employ Discretionary Access Control mechanism in that there are not many roles in our design.

## 6.3 Future Work

While the naive prototype has been developed, much needs to be done. First, the prototype's implementation now is only able to retrieve user's body-sensor data from RunKeeper. Obviously, only one data source is not enough. Since each player of TIL is equipped a Fitbit Flex wristband, it is desirable for Girji to support acquiring data from Fitbit so that TIL club is able to use Girji to analyze the players' data. Both RunKeeper and Fitbit services support OAuth protocol so that it is not difficult to retrieve data from Fitbit.

Second, function chaining should be investigated as the upload time of each input data file consumes large amount of time. After executing the function specified in the policy item, the output file resides on the OpenCPU server. Right now, we have to download the output file and upload it again as input to feed into the function in the next policy item. The round trip time leads to high end-to-end latency. The drawback can be addressed by chaining the functions of each policy item. When the next function is executed, Instead of downloading the file then uploading again, it can fetch the file by a link which is the session link of the preceding function. This way, it eliminate the necessity of downloading the file as the file is on the same server as where the next function is executed.

Third, the performance will be improved if common capabilities are cached. It is much faster to return from cache the results of the execution of some common capabilities such as the capability of retrieving a user's specific activity, fetching the data in specified time. Last, the overview of all the capabilities associated with a user should be provided to the user. With the graph of user's capabilities, better traceability can be achieved.

## 6.4 Concluding Remarks

We have presented code consent capabilities, which address the requirement of controlled sharing body-sensor data for sports analytics without giving up security. By signing the informed consent with users, analytical services are able to perform analytics on users' data with code consent capabilities. We have successfully built a prototype implementing code consent capability construction, delegation, verification as well as execution. The performance of the prototype is also evaluated. Analytical services are able to get the benefit of reuse the intermediate result by receiving delegated capabilities. Thus, the collaboration between analytical services becomes much easier to get more valuable insights from users' body-sensor data.

# A

# Informed Consent for TIL players to donate their body-sensor data

CONSENT TO PARTICIPATE IN
RELEASING BODY-SENSOR DATA FOR SPORTS ANALYTICS

• INTRODUCTION
You are asked to participate in a research study conducted by iAD lab at UiT. The research is to help the coach to get a greater level of athletes' fitness information and also make better training plans, by sending some of your body sensor data to analytical service. You are free to choose which data can be disclosed. You can withdraw the consent at any time without any penalty. You should read the information below, and ask questions about anything you do not understand, before deciding whether or not to participate.

• PROCEDURE
You voluntarily authorize some of my protected body sensor data to be used for this research. My body-sensor data includes:
    [Y] Fitness Activities.

[Y] Sleep.
[ ] Nutrition.
[Y] Weight.
[ ] Diabetes Measurements.
[Y] Calories.
[Y] Positional Data.
[Y] Heart rate.
My protected data will be kept for _1__ year(s).

• BENEFITS
To yourself, you can get recommendations about your sleepness period by disclosing your sleep data. You can also get average speed and distance in each month by disclosing your fitness activities in RunKeeper and Fitbit.

To your coach, he/she will get your ranking of performance by disclosing your position data in ZXY.

You will be reimbursed for the following out of pocket expenses that you might have, for example, you will be equipped with a Fitbit Flex to record your steps, distance, and sleep data. You will have a free premium account in RunKeeper.

• POTENTIAL RISKS AND DISCOMFORTS
If the researcher accidentally gives your data to others, you will be notified in an email which prompts if you consent the further operations.

SIGNATURE OF RESEARCH SUBJECT
I have read (or someone has read to me) the information provided above. I have been given an opportunity to ask questions and all of my questions have been answered to my satisfaction. I have been given a copy of this form.
BY SIGNING THIS FORM,
I WILLINGLY AGREE TO PARTICIPATE IN THE RESEARCH IT DESCRIBES.

___Magnus Andersen____
Name of Subject

_____ ___2013.11.02_____
Signature of Subject Date

SIGNATURE OF INVESTIGATOR

I have explained the research to the subject or his/her legal representative, and answered all of his/her questions.
I believe that he/she understands the information described in this document and freely consents to participate.

___Wei Zhang___________
Name of Investigator

_____ ____2013.11.02_____
Signature of Investigator Date

# B

# Source Code

The attached file `source_code.zip` contains Java source code for the prototype implementation packaged as projects for the Spring Tool Suite IDE.

# References


[1] Håvard D. Johansen, Wei Zhang, Joseph Hurley, and Dag Johansen. Management of body-sensor data in sports analytic with operative consent. In *Proc. of the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. IEEE, April 2014.

[2] FitnessKeeper, Inc. RunKeeper. http://runkeeper.com.

[3] NIKE, Inc. Nike+ Running. http://nikeplus.nike.com/plus/products/gps_app.

[4] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Østein Landsverk, Carsten Griwodz, Pål Halvorsen, Magnus Stenhaug, and Dag Johansen. Bagadus: An integrated real-time system for soccer analytics. *ACM Trans. Multimedia Comput. Commun. Appl.*, 10(1s):14:1–14:21, January 2014.

[5] INSTICC. *Combining Video and Player Telemetry for Evidence-Based Decisions in Soccer*. SCITEPRESS Digital Library, 2013.

[6] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[7] ZXY Sports Tracking. http://www.zxy.no.

[8] Butler W. Lampson. Protection. In *Princeton University*, pages 437–443, 1971.

[9] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[10] Ravi S. Sandhu and Pierangela Samarati. Access control: Principles and

practice. *IEEE Communications Magazine*, 32:40–48, 1994.

[11] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[12] Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos Keromytis, and Sotiris Ioannidis. Decentralized access control in distributed file systems. *ACM Comput. Surv.*, 40(3):10:1–10:30, August 2008.

[13] Tom L. Beauchamp and James Franklin Childress. *Principles of Biomedical Ethics*. Oxford University Press, 2009.

[14] Meisel A and Roth LH. What we do and do not know about informed consent. *JAMA*, 246(21):2473–2477, 1981.

[15] Deborah Estrin and Ida Sim. Open mhealth architecture: An engine for health care innovation. *Science*, 32(1):759–760, November 2010.

[16] Nithya Ramanathan, Faisal Alquaddoomi, Hossein Falaki, Dony George, C Hsieh, John Jenkins, Cameron Ketcham, Brent Longstaff, Jeroen Ooms, Joshua Selsky, et al. Ohmage: an open mobile system for activity and experience sampling. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2012 6th International Conference on*, pages 203–204. IEEE, 2012.

[17] FitnessKeeper, Inc. Health Graph API. website http://developer.runkeeper.com, November 2013.

[18] Dick Hardt. The OAuth 2.0 authorization framework. Request for Comments (RFC) 6749, Internet Engineering Task Force (IETF), October 2012.

[19] Robbert van Renesse, Håvard D. Johansen, Nihar Naigaonkar, and Dag Johansen. Secure abstraction with code capabilities. In *of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2013.

[20] Arnar Birgisson, Joe Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *of the 21th Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[21] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Request for Comments (RFC) 2104, Network Working Group, February 1997.

[22] James P. Anderson. Technical report esd-tr-73-51. *Computer Security Technology Planning Study.*, October 1972.

[23] Milestore project: A secure and scalable revocation service for codecaps.

[24] Håvard Johansen, André Allavena, and Robbert van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proc. of the 1th ACM Eurosys*, pages 3–13, 2006.

[25] Håvard D. Johansen, Dag Johansen, and Robbert van Renesse. Firepatch: Secure and time-critical dissemination of software patches. In Hein S. Venter, Mariki M. Eloff, Les Labuschagne, Jan H. P. Eloff, and Rossouw von Solms, editors, *SEC*, volume 232 of *IFIP*, pages 373–384. Springer, 2007.

[26] Jeroen Ooms. OpenCPU. website https://www.opencpu.org/.

[27] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[28] The E Language: Open Source Distributed Capabilities. http://erights.org.

[29] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[30] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[31] Michael Factor, David Nagle, Dalit Naor, Erik Riedel, and Julian Satran. The osd security protocol. In *Proceedings of the Third IEEE International Security in Storage Workshop*, SISW '05, pages 29–39, Washington, DC, USA, 2005. IEEE Computer Society.

[32] Michael Factor, Dalit Naor, Eran Rom, Julian Satran, and Sivan Tal. Capability based secure access control to networked storage devices. In *MSST*, pages 114–128. IEEE Computer Society, 2007.

[33] Andrew W. Leung, Ethan L. Miller, and Stephanie Jones. Scalable security for petascale parallel file systems. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 16:1–16:12, New

York, NY, USA, 2007. ACM.

[34] Benjamin C. Reed, Edward G. Chron, Darrell D. E. Long, and Al C. Burns. Authenticating network-attached storage. *IEEE Micro*, pages 49–57, 2000.

[35] Danny Harnik, Elliot K. Kolodner, Shahar Ronen, Julian Satran, Alexandra Shulman-Peleg, and Sivan Tal. Secure access mechanism for cloud storage. *Scalable Computing: Practice and Experience*, 12(3), 2011.

[36] J. Linn. The Kerberos Version 5 GSS-API Mechanism. RFC 1964 (Proposed Standard), June 1996. Updated by RFC 4121.

[37] Min Mun, Shuai Hao, Nilesh Mishra, Katie Shilton, Jeff Burke, Deborah Estrin, Mark Hansen, and Ramesh Govindan. Personal data vaults: A locus of control for personal data streams. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 17:1–17:12, New York, NY, USA, 2010. ACM.

[38] Mahmuda Begum, Quazi Mamun, and Mohammed Kaosar. A privacy-preserving framework for personally controlled electronic health record (pcehr) system. *2nd Australian eHealth Informatics and Security Conference*.

[39] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, August 2001.