CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Language-based permissions in embedded systems

Master's thesis in Computer Science and Engineering

MAGNUS HARRYSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Language-based permissions in embedded systems

MAGNUS HARRYSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Language-based permissions in embedded systems
MAGNUS HARRYSON

Supervisor: Alejandro Russo, Department of Computer Science and Engineering
Advisor: Mårten Söderberg, Satcube AB
Examiner: Andrei Sabelfeld, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Language-based permissions in embedded systems
MAGNUS HARRYSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Embedded devices connected to the Internet are at risk of being targeted by malware attacks that may either use the vulnerable devices as part of a bot-net or as a source of sensitive data leaks. Securing embedded devices with access control could make devices less susceptible to attacks.

This thesis investigates the usefulness and applicability of role based access control in an embedded system when using a query engine based on the logic programming language Datalog on a FreeRTOS-platform. Our findings show that implementing access control into a preexisting codebase can be done efficiently but we recognize that the query engine needs to perform the evaluation faster to meet real-time requirements. We discuss possible causes to the slow evaluation and future improvements that could increase the performance.

Keywords: Datalog, access control, FreeRTOS, RBAC, embedded system, computer science, C, macro.

Acknowledgements

I would like to give my sincerest thanks to my supervisor Alejandro Russo and my examiner Andrei Sabelfeld whose advice and guidance helped shape this thesis. I would also like to thank my company advisor Mårten Söderberg and Satcube for the opportunity to collaborate and for their invaluable feedback and assistance throughout the project.

Magnus Harryson, Gothenburg, September 2020

Contents

List of Figures

List of Tables

1
Introduction

The Internet of things is becoming more and more prevalent in our society, with smart cars, smart locks, connected thermostats, and other connected devices becoming everyday devices in our lives. Ensuring security across the IoT is a challenge and the difficulty will only increase as additional smart devices are exposed to the Internet [21].

It was reported in [1] that at one point over 500,000 devices were infected with the so-called Mirai-malware, which allowed the owner of the bot-net to shut down web-sites with denial-of-service attacks [11]. In the wake of such large-scale botnet attacks, it became evident that everyday devices connected to the Internet are at risk of being targeted by malware attacks that may either use the vulnerable devices as part of a botnet or as a source of sensitive data leaks [7]. Protection against Mirai can be easily secured by regularly updating the software and using secure configurations of usernames and passwords. Nevertheless, while protection against this specific malware can be ensured easily enough, the threat still serves as a reminder that any device connected to the Internet may be the target of attacks from malicious entities, and thus protective measures need to be taken.

To mitigate vulnerabilities, some security principles need to be considered in a system. The principle of least privilege (PoLP) is an important principle in any administrative system where multiple users require access to resources. PoLP means that each user has only the privilege[1] necessary for them to perform their designated tasks [19]. This principle is essential in reducing the number of attack vectors available to a potential attacker as well as to prevent accidental misuse among legitimate users. One can imagine an embedded system that allows multiple user connections where some functions, such as changing system configurations, should be restricted to only certain privileged administrators. Moreover, privileges and permissions are not limited to physical users. Processes within a device should also follow PoLP in order to prevent a potential rogue process from committing harmful actions within the system, or hindering a compromised process from leaking sensitive information due to being able to read restricted files.

1.1 Problem

Many IoT-devices, because of their restrictions in memory, battery, and processing power, need to make use of lightweight operating systems such as FreeRTOS. As of yet, FreeRTOS has no module or functionality for access control. As such, if a developer would want to incorporate such security in their application it would require the installation of third-party libraries, alternatively the developer could take the time to define their own

[1] Privilege in this context is permission given to an individual or group to perform an action.

implementation of access control. The first option is not always ideal as the software included may have side-effects, the library itself might be malicious, or it might introduce other attack surfaces. Furthermore, the second option would take time to implement and require expertise that the developer might not have. When faced with these possibilities, they may turn to the option of simply disregarding access control, thus compromising the device's security which can lead to data leaks or infected software. There is a gap where operating systems do not provide support for access control and so we propose to fill this gap by developing an access control module for the FreeRTOS operating system. To

1.2 Goal

The goal of this thesis is to investigate the usefulness and applicability of access control systems based on Role-Based Access Control (RBAC for short), using Datalog as the mechanism, to enforce the policy in embedded systems. To accomplish this goal, we develop a software solution that can be implemented on embedded systems using FreeRTOS as the operating system kernel. Since the aim is not just to make future systems more secure but also present ones as well, the solution will have requirements on backward compatibility and performance. Therefore, a number of requirements have been defined in Section 1.2.1 to guide the development of this project. Furthermore, we aim to have the answer to the following research questions at the end of this project:

1. Is RBAC a suitable policy to use when introducing access control in an embedded environment?

2. Can access control be done efficiently in an embedded system using RBAC and Datalog?

3. Can access control be easily introduced in an preexisting codebase?

1.2.1 Requirements

The project revolves around the solution being implementable on a system using the FreeRTOS operating system, the target system is also expected to be embedded. To guide the development of the module, a number of requirements were defined to ensure the viability of the module.

It should be impossible for a process to get illegal access to a protected action. It is essential that this requirement holds as the security hinges on the fact that the module denies illegal access requests. The other requirements as defined in the following section ensures that the module has backward compatibility and performance:

Requirement 1.
Firstly, the solution should be implementable in any eligible system, this means that the solution should not require any large modifications of existing code for it to be applicable and, as a consequence, the implementation should not require the rewriting of large amounts of code. In the context of this thesis, a solution that requires the modification of more than five lines of code per query instance will be considered a large modification.

Requirement 2.
Secondly, the developed code should not introduce any kind of severe performance degradation. If the solution produces a delay in execution, this delay should not be of major significance for the application. Therefore, an access query should take no more than 100 milliseconds to compute.

Requirement 3.
Finally, the solution should not change the inherent structure of any application implementing it. That is, the goal for the developed code is not for it to change the existing semantics but rather an additional functionality for permission checks.

As a proof of concept, the solution will be developed in collaboration with Satcube AB who develop satellite terminals on devices that share many attributes with other embedded systems, such as limited computational power, memory and battery power. As such, the case study will help determine whether the requirements are fulfilled or not. Moreover, the experience will help answering the questions defined in Section 1.1.

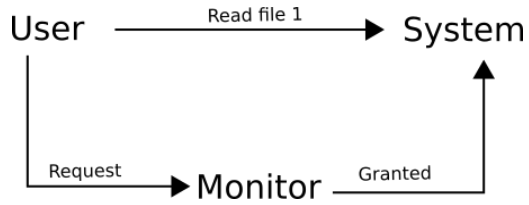
2
Background

The main purpose of access control is to protect a resource from unprivileged users. The resource needs protection as misuse can cause critical harm to the rest of the system in a computer system environment. When discussing the protected resources within a computer system, the resources are often addressed as objects and could mean things such as files, processes, and programs. Additionally, access control can be implemented in various ways such as access control lists, role-based access control, rule-based access control, etc. [9]

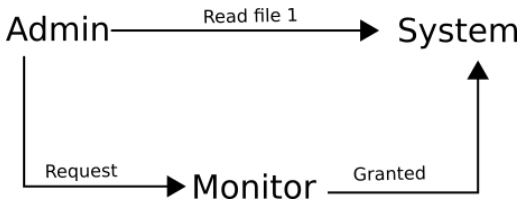
A user can be given access to certain files or actions on a device only if they have the right privilege. An example of an access control sequence can be seen in Figure 2.1 where a user wants to open and read a file on the device. The user has logged in with their credentials giving them a certain privilege and when querying the system for access will either be granted access or be denied. The monitor queries a database of some kind (depending on the implemented policy model) and returns the decision. After reading the file, the user may want to rewrite something but writing to a file requires a higher privilege, the user can instead log in with admin credentials and receive high enough privilege to change the file.

The same decisions and functionality in the previous example should apply to machine processes within a computer system. If the system is operating normally, access control should not be necessary. However, if a malfunctioning process, caused by a bug, or some other malicious software tries to change a file without being a legitimate action, some additional control is required. If the system implements access control, the process may be denied from changing the file. On the other hand, if it is not implemented, the process will be permitted to write anything to the privileged file.

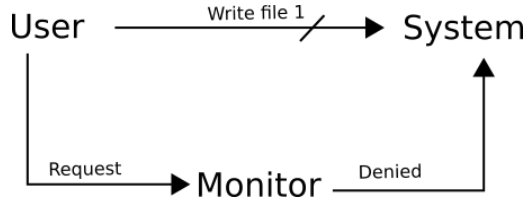
Access control list (ACL) is an example of an implementation of access control and is represented as a data structure where each entry in the list corresponds to an object access rights like: (Object - User: Operations) which gives a user permission to access an object resource with a certain permission. ACLs are implemented as tables and constitutes a simple way to uphold access rights within a system. However, as the number of users and objects get higher, the more tedious and harder it becomes to implement or maintain the list. Moreover, an ACL restricts and allows access to specific objects with certain operations. This allows for granular control but also makes the administrative cost high as each object will have multiple entries depending on the user and its user group. To review a user's or group's rights, all the ACLs need to be searched for the permissions of that user or group. [9] An example of an ACL is shown in Listing 2.1



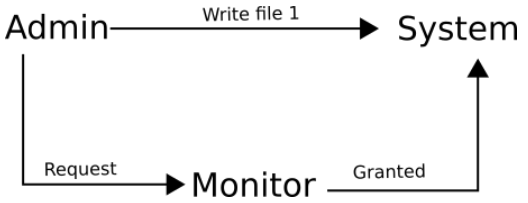
(a) The user has permission to read file 1



(b) The admin has permission to read file 1



(c) The user does not have permission to write to file 1



(d) The admin has permission to write to file 1

Figure 2.1: The figures depicts different file system scenarios in a system

```
File 1 - User: Read, Admin: Read, Write
```

Listing 2.1: Entry which could make up the ACL for a file object used to decide the scenarios in Figure 2.1

2.0.1 RBAC

Role-Based-Access-Policies (RBAC) is an access control policy model where privileges are granted to roles instead of individual users and then users are assigned to roles. When a user is assigned to a role, they are granted all privileges associated with that role. Users are actors who desire privileges to act within the system's context and can be represented as either software processes or physical users, the users are granted privileges by being members of roles. The National Institute of Standards and Technology (NIST) standard for RBAC [5] is organized into four components: core RBAC, hierarchical RBAC, static separation of duty relations, and dynamic separation of duty relations. Of the four, only core RBAC is mandatory in the NIST standard [5]. RBAC enforces PoLP by assigning just enough privileges to a role for that role to do its assigned task, which is simpler than assigning permissions to each user. Core RBAC presents the key requirements in the model presented in the list as follows:

Core RBAC requirement 1.(Many-to-many relations)
A role can have multiple privileges and a user can have multiple roles, it also means that a user can have multiple roles and a role can have multiple users.

Core RBAC requirement 2.(User-role review)
The role assignments of a user and the users assigned to a role can be easily determined. This also includes an advanced review function in reviewing the role-privilege relation.

Core RBAC requirement 3.(User sessions)

User sessions allow selective activation of role privileges. During a session, a user may activate a role and thus gain all the accompanying privileges. Each session is associated with a user and each user is associated with one or more sessions.

Core RBAC requirement 4.(Multiple role activation)
This requirement allows users to activate multiple roles and their privileges at the same time.

There exist other versions of the RBAC with parts and features that have made its way into the NIST standard [18, 20]. As such, the standard is a well-formed version of RBAC, the same version was adopted as an American National Standards Institute standard in 2004 [17]. However, there are also critiques against this adopted standard brought forth by Bertino et al. [12]. In their paper, they make several suggestions to improve the standard, such as removing the notion of sessions from core RBAC and introducing it as a separate component.

Privileges are defined as the right to do some protected operation on objects(OBS). In the RBAC standard, an object in the context of computer systems can mean information containers such as files or directories, it could also represent other types of objects relevant to a system such as memory or disk space. Operations(OPS), in the same context, could be the reading of a file or memory allocation.

In Listing 2.2 a list is used to represent the RBAC way of handling access control, each entry represent an assignment of user or privilege. The lines 1 and 2 represent the assignments of *User* to a default role and *Admin* to a privileged role. In lines 3-5, permissions to use the operations *File read* and *File write* are granted to roles.

```
1 User, Normal role
2 Admin, Privileged role
3 File read, File 1, Normal role
4 File read, File 1, Privileged role
5 File write, File 1, Privileged role
```

Listing 2.2: RBAC styled access control used to model the scenarios in Figure 2.1

Hierarchical RBAC is a component of standard RBAC and offers a way for roles to inherit permissions of other roles. By using role inheritance, it is possible to build a structure of role dependencies such that privileges does not need to be assigned to multiple roles when they have common subsets. Listing 2.2 can be adjusted to make the privileged role in the example inherit the privileges owned by the normal role, the result is shown in Listing 2.3 where on line 5 the *Privileged role* inherits privileges from the *Normal role*.

```
1 User, Normal role
2 Admin, Privileged role
3 File read, File 1, Normal role
4 File write, File 1, Privileged role
5 Privileged role, Normal role
```

Listing 2.3: Hierarchical RBAC styled access control used to model the scenarios in Figure 2.1

2.1 FreeRTOS

FreeRTOS is an open-source real-time operating system kernel designed to run effectively on micro-controllers and as such is typically used in embedded systems[6]. An application using a real-time operating system (RTOS) kernel can be seen as a structure of independent tasks that each runs in their own context. A task has no dependencies towards the other tasks running nor the scheduler. If the processor running the application has only one core, which is common in smaller systems, only one task can be run at any time. It is the job of the scheduler to start and stop each task as the application executes. As a consequence, it is entirely up to the application writer to create a task that handles access control or write their application in such a way that PoLP is always fulfilled. This section will cover some of the functionalities FreeRTOS offer; it is expected that the reader has some basic knowledge of process management, operating systems, and programming in the C-language. Not every function and concept from the FreeRTOS architecture will be covered here so for full coverage of FreeRTOS and its API, readers are referred to FreeRTOS' webpage [6]

2.1.1 Task creation

Tasks in FreeRTOS can be created with either *xTaskCreate()* or *xTaskCreateStatic()*. In the former, memory is automatically allocated from the FreeRTOS heap whereas in the second, the application writer has to provide the RAM. The static function requires two additional parameters defining the memory, the other parameters are described in the following section.

```
xTaskCreate(pvTaskCode,pcName,usStackDepth,pvParameters,uxPriority,
            pxCreatedTask)
```

pvTaskCode.
The task is a C function that usually runs in an infinite loop or until it is deleted by itself. This parameter is a pointer to a custom function that executes the desired task.

pcName.
The task is provided with a hard-coded string that is assigned to the task, this name is rarely used in the FreeRTOS API but is a way for developers to debug or obtain the task handle using the function *xTaskGetHandle()*.

usStackDepth.
Each task has its own unique stack that is allocated when the task is created, this value tells the kernel how large to make the stack. The minimum stack size is defined in the application-defined FreeRTOS config[1]. The argument is of the type uint32_t.

pvParameters.
A value to be passed to the function implementing the task, this parameter takes the type of a void pointer. As such, any value can be passed to the function task.

uxPriority.
This value decides what priority the task has in the scheduler. This value can be any value

[1]The FreeRTOS config is the header file in which most of FreeRTOS configurations lie, FreeRTOS.h

between 0 (the lowest) and (MAX_PRIORITIES-1) (the highest), configMAX_PRIORITIES is defined in the FreeRTOS config.

pxCreatedTask.
An optional value that hands out a task handle that later can be used to lower the task's priority or delete the task. If the handle is undesired, the value can be set to NULL.

2.1.2 Task states

After a task has been created it enters the so-called **Running state** where it is allowed to run as long as it is the task with the highest set priority in that state. This presents a problem where every task needs the same priority or the task with the highest priority will run continuously without allowing any other task to run. The solution is to have an event-driven system where the tasks enter a blocked state while waiting for some event to trigger it.

Overall, there exist four states in which a task can live. The **Running state** where a task is executed by the processor. In a processor with a single core, only one task can be executed at a time. This state is not to be confused with the number of tasks waiting to be executed, if a task is not being executed it will enter a **Ready state** in which it waits for its turn either by having the previous task finish executing or a change in priorities. If multiple tasks have the same level of priority, the scheduler will execute them in round-robin style.

The **Blocked state** is the state a task enters when it waits for an event to occur. There are typically two types of events; temporal events meaning a delay period where the task is waiting a certain amount of time before returning to the Running state. The second type of event is synchronization events where the task waits for events that originate from another task or interrupt. These could mean messages passed via a queue or semaphore-signals. Moreover, waiting for a temporal event does not inhibit the task from reacting to a synchronization event.

The final state is the **Suspended state** where any existing task is unavailable to the scheduler. The only way in or out of this state is by calling the API functions *vTaskSuspend()* and *vTaskResume()*. Figure 2.2 depicts the transition diagram of the tasks.

2.1.3 Message passing

At certain times during execution, the tasks need to communicate and FreeRTOS allows such functionality with the help of queues. Similar to the task creation functions, the queues can be created statically where the application writer has to provide the RAM.

xQueueCreate() has two parameters, the first is the value which defines how many items the queue can hold and the second parameter sets the size of each item in the queue. The function returns a NULL value if it fails or a handle by which another task can pass messages.

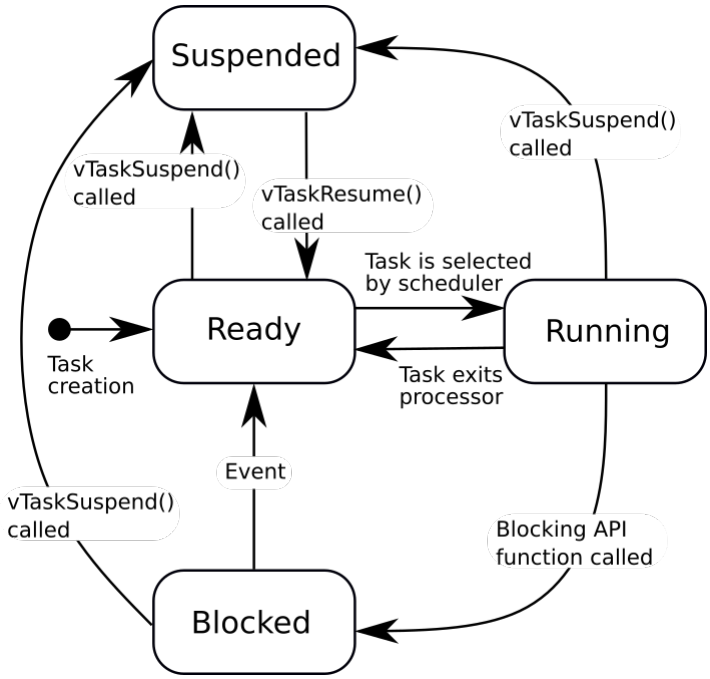


Figure 2.2: Transition diagram of the different states available from the FreeRTOS API

xQueueRecieve() is the function which reads the item from the queue, it has three parameters. The first parameter is the handle supplied which was created by *xQueueCreate*, the second is a pointer to the memory into which the data will be copied. The last parameter determines how long the task should wait in the Blocked state to receive items from the queue. The result is either pdPASS if data was successfully read from the queue or an error if there are no items in the queue.

xQueueSend() is used to send data via the queue, it has the same parameters as *xQueueRecieve*.

2.1.4 Synchronization

When handling multiple tasks and shared resources, it is important to make sure there exist no ways to cause race conditions which can produce unintended behavior and effects. Moreover, sometimes tasks need to synchronize. As such, FreeRTOS has support for semaphores and mutexes which come in different implementations covered in this section.

Counting semaphores.
Counting semaphores are typically used in two different ways, the first way is as a form of event counter. The task will 'give' the semaphore when an event occurs and then 'take' it back when the event has been processed. This produces a counter which represents the difference between the number of events occurred and the number of events processed.
The second way of using counting semaphores is in resource management, a task will need access to a resource and thus will try to 'take' the semaphore decrementing the counter

associated with the semaphore. If the counter is zero, the task will enter the Blocked state and wait until it is signaled by another task that is finished with the resource 'giving' back the semaphore which increments the counter. The counting semaphore can be initialized so that it functions as a binary semaphore.

Binary semaphores and mutexes.
Binary semaphores can be used for synchronization and if used this way the 'taken' semaphore does not need to be 'given' back as the other task will 'give' the semaphore when it is ready to synchronize.
Mutexes are a special kind of binary semaphores used to control access to some shared resource. When a task wants to access a shared resource, it tries to 'take' the mutex and if it succeeds it will be allowed to continue. If the task fails to 'take' the mutex, it enters the Blocked state and will wait until the mutex is available. The mutex needs to be 'given' back as to not forever block the mutex and resource for the other tasks.

2.2 Logic programming and Datalog

Logic programming is a programming paradigm based largely on formal logic. Logic programming is an appreciated alternative to other programming paradigms because using first-order logic to form policies allow them to have clear syntax and semantics. Logic programming is therefore regarded as a suitable way to solve access control problems [8]. A program written in a logic programming language will have the structure of a set of clauses expressing facts. The evaluation of a logic program is designed to answer a query or derive additional truths about the set of statements in the program. The most prominent programming languages are Prolog, answer set programming(ASP), and Datalog. In each of these, sentences are structured as clauses in the form: H :- B_1,...,B_n where for H to be true, all B_1,...,B_n are required to be true also.

The evaluation of logic programs is typically done in one of two different strategies: the top-down approach or the bottom-up approach. The strategy of bottom-up evaluation is to start from all facts in a program to derive new facts and as such producing all consequences of the given program. However, when querying for a single answer, this approach becomes wasteful. The alternative, the top-down approach, is to solve for a single fact which will produce additional facts to be solved. This process will continue until no more facts are produced resulting in a collection of facts supporting the initial query. Therefore when querying for one answer, the top-down approach seems most relevant.

2.2.1 DATALOG

Datalog is a declarative logic programming language that uses the notion of first-order logic combined with database theory in an effort to make querying easier and more efficient. Datalog has seen previous use in the access control field [2, 4] and has been extended in various ways [13, 15] to fill the gaps of its functionality. Moreover, the language has been studied extensively in the field of programming languages. These facts along with its tractability make the language a good candidate for use in embedded environments. This section will cover what is needed to know about Datalog to understand

its implementation as the access control mechanism.

A DATALOG program is made up out of a set of Horn clauses with the following representation: L_0 :- L_1,...,L_n, where each literal L_i is of the form $p_i(t_1,...,t_{k_i})$ in which p_i is a predicate and t_k is a term. A term can be either a constant or a variable. The left-hand side of the clause is called the head and the right-hand side is called the body, a clause without a body is called a fact and rule otherwise. For the left-hand side to be true in a DATALOG rule, all the predicates in the right-hand side need to be evaluated to true.

```
1 parent(john,bob).
2 parent(bob,alice).
3 grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

Listing 2.4: A small DATALOG program

In Listing 2.4, querying for any grandparents with the literal $grandparent(X, Y)$ would result in the tuple <john, alice> which in normal text form would equate to "John is a parent to Bob who is a parent to Alice and therefore John is a grandparent to Alice".

In DATALOG, terms beginning with capital letters are treated as variables and if they start with a lower-case letter they are treated as constants. As such, in the example above, X, Y and, Z are variables, *john*, *bob*, and *alice* are constants whereas *grandparent* and *parent* are predicates. Moreover, any variable occurring in the head of a rule must also occur in the body of the same rule. It follows that all facts are without variables and a literal without variables is called ground. All literals with the same predicate symbol are required to keep the same amount of arguments(arity), these presented conditions guarantee that all rules derivable from a DATALOG program are finite.

The example in Figure 2.1 can be modeled in ACL and RBAC using DATALOG as the query engine as can be seen in Listing 2.5 and 2.6. To query the authorization for a process called *user* and the action *file write* for some file *file_1* one can supply the DATALOG engine with the query *authorized(file_write, file_1, user)* to get all the tuples that match the given query. In the example, such a query will produce no results, however replacing the term *file_write* with *file_read* will produce the tuple <file_read, file_1, user>. From the listings, it's apparent that it requires fewer lines for the same functionality in the ACL model. On the other hand, if more actors are added who need the same privileges as admin, the ACL strategy would require two entries per actor while in RBAC only one entry is needed.

```
1 authorized(file_read, file_1, user).
2 authorized(file_read, file_1, admin).
3 authorized(file_write, file_1 admin).
```

Listing 2.5: A DATALOG program modeled after ACL from which the scenario in Figure 2.1 can be derived

```
1 member(user, normal_role).
2 member(admin, privileged_role).
3 privilege(file_read, file_1, normal_role).
4 privilege(file_read, file_1, privileged_role).
5 privilege(file_write, file_1, privileged_role).
6 authorized(X, F, Y) :- member(Y,Z), privilege(X, F, Z).
```

Listing 2.6: A DATALOG program modeled after RBAC from which the scenario in Figure 2.1 can be derived

3
Implementation

The project consisted of several parts that needed to be developed concurrently: configuring a suitable RBAC policy and its implementation in Datalog, creating a monitor in C programming language with the FreeRTOS API, and finally, find a way to hook the access control module into the existing Satcube-codebase. A flowchart of the execution of a process can be seen in Figure 3.1 where the access control module responds to a request made from the adaptation layer. The adaption layer is, as the name suggests, a layer in which all the necessary functions from the included libraries have been adapted and simplified to serve the application's purpose. The monitor serves as the entry point for the request and after processing the request, it queries the Datalog-engine if the calling process has permission to access the requested action. The Datalog program in turn was written and modeled after RBAC. When the monitor responds with either *allow* or *deny*, the process either proceeds to execute the called code if the response is *allow* or return the value associated with access denial if the request was *denied*.

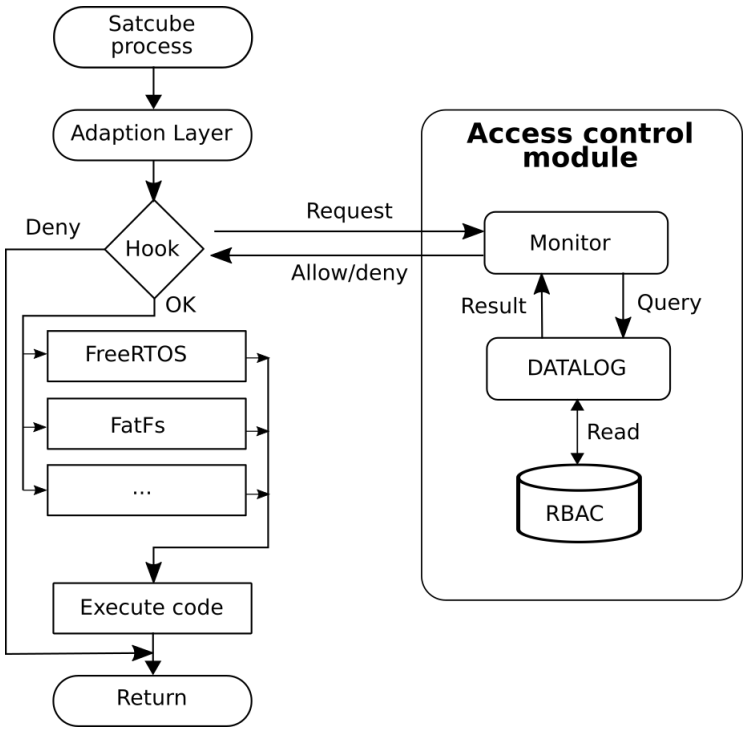


Figure 3.1: Execution of a process with hooks in the adaption layer.

The integration with Satcube's developed C-code and its target hardware STM32F769I-DISCOVERY[1] served as the first stage of technical development. As described in Section 1.1, embedded-devices are often restricted in memory and so our target hardware needed to be one with limited resources and as such STM32F769IDISCOVERY is an attractive target board, based on a micro-controller using the ARM® Cortex®M7 core. The board uses around 17 Mbytes of RAM and offers the possibility to add an SD-card for additional storage and while this makes it quite powerful, it is still a board of smaller size and capability compared to other boards.

As Figure 3.1 suggests, we aimed to first apply the access control to Satcube's adaption layer to have an easy entry point for the access control and an environment that we knew had support in the existing codebase. After having successfully implemented the access control for the adaption layer, the attempt was replicated but focused on making a more generalized approach to the protected library-functions as can be seen in Figure 3.2.

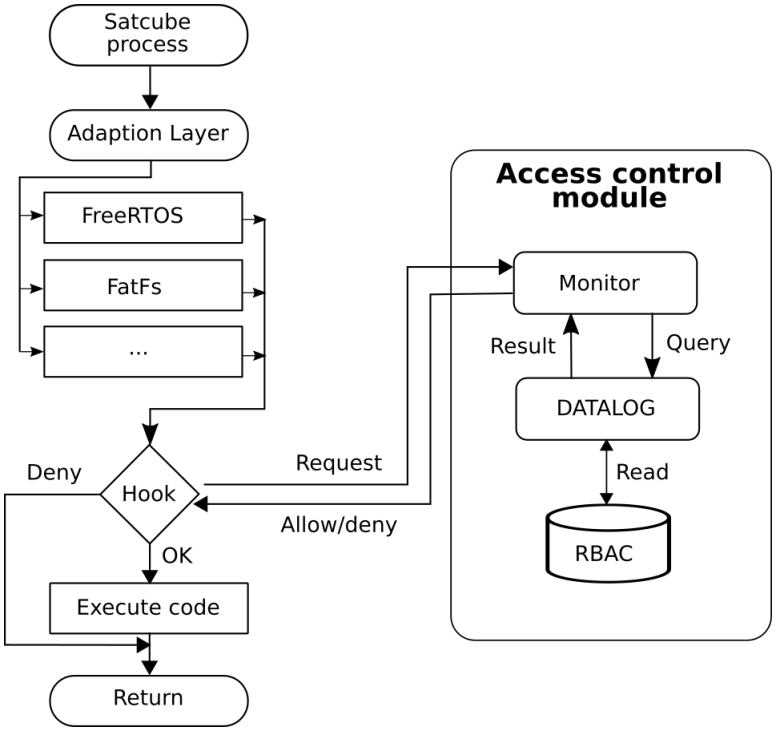


Figure 3.2: Execution of a process with hooks directly into the libraries.

3.1 Technical Implementation

The purpose of this section is to describe how the technical implementation is structured and how the FreeRTOS API is used. The section will also cover any limitations or dependencies of the solution. For the access control to be able to initialize the Datalog database, the system needs to first start a thread(called a task in the FreeRTOS environment) within which the access control will run. As such, every time a process requests access to perform an action, the access control-task will be messaged and a check will be

[1]https://www.st.com/en/evaluation-tools/32f769idiscovery.html

made.

3.1.1 FreeRTOS initialization

From Listing 3.1 we can follow the initialization of the FreeRTOS task implemented via the API defined in Section 2.1. A message queue is created on line 1 which will handle the communication between the running tasks. If the queue is created successfully the execution will proceed to create a mutex-object as represented on line 6, the mutex will act as the gate such that multiple tasks cannot simultaneously request access as that might introduce race conditions.

The same procedure is repeated where if we succeed, we create a semaphore initialized to 0 on line 12. The semaphore will be used to synchronize the tasks. As such, the semaphore will be initialized as binary as described in Section 2.1. Finally with the task's environment set up, we create the task on line 18. Note however that the task's priority, on line 23, should be high enough such that high-priority tasks are not in a blocked state because there is another task with a higher priority than the access control task running effectively blocking the access control task from evaluating the request. The parameter *accessControlTask* on line 19 defines the function which will be the entry-point for the newly created task which will be further discussed later in this section. Note that in-between each FreeRTOS function, there is a specific check to see if the handles have been created correctly.

```
1  queueHandle = xQueueCreate(ACCESS_CONTROL_QUEUE_SIZE, sizeof(char *));
2  if (queueHandle == NULL) {
3    result = AC_FAILED;
4  }
5  if (result == AC_OK) {
6    requestsPendingMutex = xSemaphoreCreateRecursiveMutex();
7    if (requestsPendingMutex == NULL) {
8      result = AC_FAILED;
9    }
10 }
11 if (result == AC_OK){
12   returnDlSemaphore = xSemaphoreCreateCounting(1, 0);
13   if (returnDlSemaphore == NULL) {
14     result = AC_FAILED;
15   }
16 }
17 if (result == AC_OK) {
18   rtosResult = xTaskCreate(
19     accessControlTask,
20     "Access␣Control",
21     ACCESS_CONTROL_STACK_SIZE,
22     NULL,
23     ACCESS_CONTROL_TASK_PRIO,
24     NULL
25   );
26   if (rtosResult != pdPASS) {
27     result = AC_FAILED;
28   }
```

```
29 }
```

Listing 3.1: Initialization of the access control task

3.1.2 Datalog initialization and querying

There are several open-source implementations of DATALOG available for use. However, not all of these are suitable for an embedded system. XSB,[2] for instance is an extensive logic programming and deductive database system for Unix and Windows but even though XSB uses DATALOG as its logic engine, its size makes it cumbersome and difficult to use in an embedded system. Furthermore, it lacks the support to be installed on our target system.

We decided to use the DATALOG package[3] which is a lightweight implementation of a DATALOG interpreter running on a Lua stack. It also includes an API with support for C implementation. Lua uses functions and libraries that may not always be available in embedded devices. As a result, a custom open-source Lua-library called *embLua*[4] designed specifically for C-developers coding for embedded devices was used in place of the originally included Lua-library. Both the target system provided by Satcube and embLua uses a file system-library called *FatFs*[5] which similarly to embLua is developed specifically for embedded systems. The developed solution, therefore uses this file system and as such the final DATALOG package used to implement the access control has dependencies in embLua and FatFs.

The initialization phase of the module expects one or more DATALOG programs from which database structures can be initialized. The system is set up to allow multiple databases, allowing for different sets of privileges at different times. This can be beneficial for the application to allow a certain initialization process to read files. For instance, if a task needs to read its configuration files at its initialization phase, it should be able to do so even if it is not permitted to read files at default.

It follows from the definitions made in Section 2.1 that the task should run without exiting. This is solved in C by implementing a so-called for-loop without conditions, resulting in an infinite loop. When the database has been initialized with whatever program the user-specified, the task enters a Blocked state to wait for an access request which can be seen on line 2 in Listing 3.2. When the task receives a message, it is put in the Running or Ready state. The message is checked to see what action is to be performed and executes the function as can be seen on line 3 through 8.

```
1 static void accessControlTask(){
2   for( ;; ) {
3     xQueueReceive(queueHandle, &msg, ACCESS_CONTROL_TIMEOUT_MAX)
4     switch(msg) {
5       case ACCESS_CONTROL_MSG_REQUEST:
6         constructDatalogQuery(username,action);
7       case ACCESS_CONTROL_MSG_DATABASE:
8         loadDatabase(filename);
```

[2] http://xsb.sourceforge.net/index.html
[3] https://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html
[4] https://github.com/szieke/embLua
[5] http://elm-chan.org/fsw/ff/00index_e.html

```
 9       }
10     }
11  }
```

Listing 3.2: The access control task is placed in the Blocked state waiting for messages inside a neverending loop

The *constructDatalogQuery* function on line 6 makes use of two variables, the user requesting access and the action the user wants access to. The function constructs a query using the API made available by the DATALOG package. The query is constructed by pushing parts of the query, piece by piece, on a Lua-stack. To showcase the construction of a query, we formulate the arbitrary sentence: is user *Ethernet* a member of the role *Root*. This query corresponds to *member("Ethernet","Root")* in DATALOG and the equivalent procedure in Listing 3.3.

```
1  // Begin constructing a literal, "db" is the initialized database
2  dl_pushliteral(db);
3  // Write a string to the stack
4  dl_pushstring(db,"member");
5  // Use the string as a predicate
6  dl_addpred(db);                    // member(
7  //Add two terms to the predicate
8  dl_pushstring(db,"Ethernet");
9  dl_addconst(db);                   // member("Ethernet"
10 dl_pushstring(db,"Root");
11 dl_addconst(db);                   // member("Ethernet","Root"
12 // Finish the literal
13 dl_makeliteral(db);                // member("Ethernet","Root")
14 dl_answers_t answers;
15 // Query the database using the created literal
16 dl_ask(db,&answers);               // member(Ethernet","Root")?
```

Listing 3.3: C-code showing how the DATALOG API is used to create datalog-queries.

The procedure begins by starting a literal, this is followed by creating the predicate "member" through lines 4 and 6. "Ethernet" and "Root" are made into the literal's terms on lines 8 through 11 after which the literal is finished and queried against the database *db*. The result is saved in the variable *answers* defined on line 14. If *answers* is NULL, no answers were found and we can conclude that *Ethernet* is not a member of *Root*. Otherwise the variable contains a structure with the literal *member("Ethernet","Root")* on it.

3.1.3 Access control wrapper

Adhering to requirements 1 and 2 from Section 1.2.1, the bridge between application and access control module needed to be easily implementable and not change any existing semantics. In the C programming language, it is possible to use pre-processor arguments (or macros as they are commonly called). Macros allow us to define a fragment of code with a certain name. When the name is called, the code it represents is inserted in the place of the name. Listings 3.4 and 3.5 show examples of how macros can be defined to exercise access control in an application.

```
1 #define ACCESS_CONTROL_CALL(actionName,action) \
2    checkAccess(actionName) ? AC_NOT_AUTHORIZED : action
```

Listing 3.4: An example of usage of macros in C

The macro in Listing 3.4 is wrapped around the action which requires permission to execute, with the name of the function as the first argument and the function itself as the second. The function *checkAccess* returns zero on success and a non-zero value on failure which will result in the execution of the action if successful and a predefined error-value if not. This macro allows us to decide what functions and thereby what actions should require permissions to access.

```
1 #undef f_read
2 #define f_read(args...) \
3     checkAccess("f_read" ? AC_NOT_AUTHORIZED : \
4     f_read(args)
```

Listing 3.5: An example of usage of macros in C

The macro in Listing 3.5 works a little differently than from the one in Listing 3.4. Each entry made this way corresponds to a protected function in the desired policy. On line 1, the function is undefined so that on line 2, we can redefine it so that it is gated by the *checkAccess* function. Instead of wrapping each function as we did with the first macro, this version will replace any instance of *f_read*. First, controlling its access, and then if authorized use the original *f_read*. This works because macros are non-recursive, meaning it will not redefine itself in an endless loop as it will only do it once, calling the original *f_read* the second time.

3.1.4 Limitations

While many different variations with various benefits exist, we chose a Datalog package that uses a simple variation of the language. As a result, the solution will not be able to use more advanced functionalities as the ones described in [13, 15]. However, this is not a restriction to our case-study as the module relies only on the fundamental functionality of Datalog. Li and Mitchell make use of Datalog extended with constraints in [13] which allows them to use Datalog in constraint domains, which supports the use first-order formulas in file hierarchies and time constraints. We see the potential upside of this functionality but it is not functionality that is vital for access control. In [15], the authors make use of Datalog with negation in order to express exceptions in hierarchical RBAC which could provide support in defining a more advanced policy program.

3.2 Policy

When planning an access control system, three points should be considered: policy, model, and mechanisms. The policy is a high-level requirement that details who should be able to access what and under what circumstances. Mechanisms are used to enforce the policy, by either denying or permitting access within the system. Examples of such mechanisms are table lookups, database queries, or logic programs like Datalog. Finally, a model is a formal representation of the policy and can be seen as the bridge between the policy and the mechanism [9].

3.2.1 RBAC

In this thesis, objects have been combined with the operation in favor of making a simpler program. As such, instead of having separate mappings between actions(ACTS) and

objects(OBS) we can instead have it implied implicitly that if the user has the right to perform the action, it does not matter on which object the action is made. This makes it easier to use the DATALOG package which has difficulties defining file structures and therefore makes it cumbersome to define control over files. As such, instead of having ACTS and OBS as different sets, there is now only ACTS. Furthermore, it was presented in Section 2 that sessions in RBAC are not necessary for RBAC to function properly, it is therefore decided that it is not needed in the scope of what this project tries to accomplish.

With the session requirement disregarded and using the custom privilege definition, this thesis' core RBAC's definition is as follows:

Definition 1
- USERS, ROLES, ACTS represents the sets containing all users, roles, and actions respectively.

- UA $\subseteq$ *USERS* x *ROLES represents the set containing a many-to-many mapping between users and roles.*

- PA $\subseteq$ *ACTS* x *ROLES represents the set of a many-to-many mapping between actions and roles.*

Hierarchical RBAC allows the inclusion of role inheritance with the following definition:

Definition 2
- RH $\subseteq$ *ROLES* x *ROLES, RH*(r_1, r_2)*, is a partial order* $r_1 \succeq r_2$ *where users who are members of* r_1 *are also members of* r_2 *and all privileges assigned to* r_2 *is also assigned to* r_1*.*

The hierarchical relations defined in Definition 2 expands the set UA from Definition 1 to also include a mapping from the user u to role r_2 if there exists a mapping from u to role r_1 and if $RH(r_1, r_2)$. Given these definitions of RBAC sets, we can further define functions operating on the defined sets:

Definition 3
- assigned_users($u \in$ *USERS)* $\rightarrow 2^{ROLES}$ *returns the roles which user u is assigned to, formally:* $\{r \in ROLES; (u \mapsto r) \in UA\}$

- assigned_privileges($a \in$ *ACTS)* $\rightarrow 2^{ROLES}$ *returns the roles which have the which possesses the privilege to use action a, formally:* $\{r \in ROLES; (r \mapsto a) \in PA\}$

- user_privilege($u \in$ *USERS,* $a \in$ *ACTS) returns either the empty set* $\emptyset$*, or the set Q, formally: {Q = assigned_users(u)* $\cap$ *assigned_ privileges(a)}*

The *assigned_users(u)* function allows the querying for all roles which are assigned to the user u, with the conditions that role r needs to be a member of the set ROLES, user u is a member of the set USERS, and there must exist a mapping from u to r in the set UA. Similarly in the function *assigned_privileges(a)*, all roles associated with action a are returned, with the conditions that role r is a member of the set ROLES, action a is a member of the set ACTS, and there must exist a mapping from r to a in the set PA. Finally, the function *user_privilege()* provides a query which returns the intersection of

the result of *assigned_users()* and *assigned_privileges()*, where user u is a member of the set USERS and a is a member of set ACTS.

3.2.2 Datalog as the policy mechanism

DATALOG is used as the policy mechanism to evaluate the permissions assigned to each user in the RBAC policy. DATALOG can infer new rules from the ones specified in the original policy making it possible to define a policy language in a compressed way. In DATALOG we do not need to define sets of objects before making use of them as terms in the resulting program. As such, the second item in Definition 1 can be defined in DATALOG without explicitly defining the sets. If, for instance, *Ethernet* is a user and *Root* is a role then *Ethernet* can be added to *Root* by defining a DATALOG-rule like so: *member("Ethernet", "Root")*. This rule now equals a mapping from the user *Ethernet* to the role *Root* and is correct even though *Ethernet* and *Root* have yet to be defined as user and role.

However, individual sets are necessary for the enabling of review functions mentioned in Section 3.2.1. Furthermore, it is an essential requirement that each variable sentenced on the left-hand side of the rule is also included on the right-hand side. This requirement allows the evaluation of a DATALOG program to be finite and for the deductive functionality to work as intended. If we were to implement a rule that lets a user, *Ethernet*, be a member of all roles, we could define it like so: *member("Ethernet", X)*. The variable X acts as a wildcard and so DATALOG will place any constants in its stead. However, this renders the fact unsafe as X can be evaluated as anything, including another user which would result in undefined behavior since a user can not be a member of another user. Instead, the rule should be formulated as such: member("Ethernet", X) :- role(X). The right-hand side of the rule forces X to be a role, rendering the rule safe for use. It is therefore crucial that enough ground-cases are included for the program to function properly. As a result, to able to make use of variables in rules in DATALOG we must include the base-cases (the individual sets) in the DATALOG program.

```
1  % ROLES set %
2  role("Root").
3  role("Fsread").
4  role("Fswrite").
5  role("Kernel").
6
7  % USERS set %
8  user("MbedWeb␣Main").
9  user("Web␣WT").
10 user("Ethernet").
11 user("System␣Control").
12
13 % ACTS set %
14 action("f_open").
15 action("f_write").
16 action("f_close").
17 action("f_read").
18 action("xTaskCreate").
19 action("vTaskDelete").
20 action("xSemaphoreCreateCounting").
```

```
21 action("xQueueCreate").
```

Listing 3.6: Declaration of the base classes.

The program in Listing 3.6 is a snippet from the DATALOG program with which access control is derived. The facts form the sets: USERS, ROLES, and ACTS defined in Definition 1. In this context, the constant in the user literal, e.g. "VNC Server", is the task name assigned to a task by the function *xTaskCreate()*. The task names are accessible with the function *pcTaskGetName()*) and the constants in the action literals, e.g. *f_write*, are implemented functions from the application.

From the code snippet, two observations can be made: firstly, the implemented DATALOG package allows for constants to be defined as strings with the use of quotation marks. This fact makes the limits on whitespace and constants beginning with lower-case letters less restrictive. The second observation is that while the constants within the role literal can be any arbitrary string defined by the application writer, the constants in the user and action literals need to be the same as their counterpart in the FreeRTOS system and the application's codebase. From the observations it's clear that the labeling of actions and functions in DATALOG becomes rigid when it's mandated by the application and its implementation. However, this is easily surmountable given that strings are allowed as parameters in this particular DATALOG package.

```
1  % user-role  mapping, UA %
2  member("Ethernet","Root").
3  member("Web␣WT","Task").
4  member("Web␣WT","Fsread").
5  member("Web␣Main","Kernel").
6
7  % role-action mapping, PA %
8  privilege("Task","xTaskCreate").
9  privilege("Task","vTaskDelete").
10 privilege("Kernel","xSemaphoreCreateCounting").
11 privilege("Kernel","xQueueCreate").
12 privilege("Fsread","f_read").
13 privilege("Fswrite","f_write").
14
15 % "Root" is allowed to do everything %
16 privilege("Root", X) :- action(X).
17
18 % Roles inheritance, RH %
19 directInherit("Kernel","Task").
20 directInherit("Fswrite","Fsread").
```

Listing 3.7: Role and privilege assignment.

```
1 %--- Inherit Privilege ---%
2 inherit(A,B) :- directInherit(A,B).
3 inherit(A,B) :- directInherit(A,C), inherit(C,B).
4 member(A,C) :- inherit(B,C), member(A,B).
5
6 % user_privilege = assigned_users ∩ assigned_privileges %
7 authorized(A,B) :- member(A,C), privilege(C,B).
```

Listing 3.8: Role inheritance rules and the rule from which all permissions are derived.

In Listing 3.7, the users are assigned memberships in roles on lines 2-5. Privileges are assigned to the roles on lines 8-13. Note that on line 16, we define "Root" to be allowed to do everything with the help of variable X. Finally, the rule which all access checks will be queried with: *authorized(A,B) :- member(A,C), privilege(C,B).* The rule will find all tuples satisfying the criteria specified in the query, for example, if the query were to be "authorized("Ethernet","f_read")?" it would result in the tuple: <"Ethernet","f_read"> meaning the user "Ethernet" has the privilege to use the function "f_read".

4
Evaluation

In this chapter, we evaluate the access control module based on the requirements defined in Section 1.2.1. The focus of the evaluation has been on determining if controlling access to a set of FreeRTOS functions is done semantically correct and if the performance is not severely worsened. Furthermore, we also tested a set of file system functions in FatFs that reflect the functionality included in FreeRTOS' file system library. The reason for using FatFs is to allow testing on Satcube's system and codebase which uses this open-source library. The set of functions included are presented in Table 4.1a where we can see that the actions have an equivalent option in the corresponding library. Tests were conducted that evaluated whether access controlling the kernel-functions such as synchronization and message passing could be done efficiently and without introducing semantic issues. An example set of available functions in the two libraries can be seen in Table 4.1

FatFs		FreeRTOS	
f_mkdir	f_open	ff_mkdir	ff_fopen
f_write	f_close	ff_fwrite	ff_fclose
f_unlink	f_read	ff_remove	ff_fread
f_state	f_lseek	ff_stat	ff_fseek

(a) FatFs' filesystem functions and the corresponding FreeRTOS' functions.

FreeRTOS Kernel
xTaskCreate
vTaskDelete
xSemaphoreCreateCounting
xQueueCreate

(b) A subset of FreeRTOS' kernel functions

Table 4.1: An example set of two libraries' functions

4.1 Semantics

This section covers how well the solution fulfills the first and third requirements from Section 1.2.1. Since requirement 3 is not measured by value, the results will be made from presenting the code and arguing for or against if the requirement has been fulfilled.

```
#define ACCESS_CONTROL_CALL(actionName,action) \
   checkAccess(actionName) ? AC_NOT_AUTHORIZED : action
```

Listing 4.1: Macro code from Listing 3.4

With the use of macros, we smoothly replace preexisting code with our own code. Example usage of the macro from Listing 4.1 could be the following:

```
fileResult = ACCESS_CONTROL_CALL("f_read",f_read(args...));
```

Which would upon compilation expand into:

```
if(accessCheck("f_read"))
  fileResult = AC_NOT_AUTHORIZED;
else
  fileResult = f_read(args);
```

As can be seen, the only addition to the original code is the access check and an *if-statement*, resolving whether the returned result should be an error-value on failure or the execution of the protected function and its return value on success. However, not every function has the same return value on failure. For instance, if *xQueueCreate* fails to create a queue, it will return a NULL value equaling zero, while *f_read* returns a non-zero value on failure. A wrongfully defined value would cause the application to believe the operation was successful even though it failed. Although this prevents the usage of a single value for all failures, macros are still usable but put more responsibility on the application writer to configure values such that they match the expected value for failure. Apart from this condition, the semantics of the code execution remains the same.

```
#undef f_read
#define f_read(args...) \
   checkAccess("f_read" ? AC_NOT_AUTHORIZED : \
   f_read(args)
```

Listing 4.2: Macro code from Listing 3.5

In the second version of the macro, the one defined in Listing 4.2, we redefine the protected function. This happens automatically with the inclusion of the header file in which the macro is defined so using the previous example we would get have the following sequence:

```
fileResult = f_read(args...);
```

Which would upon compilation expand into:

```
if(accessCheck("f_read"))
  fileResult = AC_NOT_AUTHORIZED;
else
  fileResult = f_read(args);
```

The macros produce identical results, but the latter has the benefit of the absence of the macro-wrapper making the code cleaner. The preparations are however more extensive as each protected function needs its specific redefinition as can be seen in Listing 4.2. This fact makes this macro more tedious to use as the size of the header file will have a linear relation to the number of protected functions. A final note of importance is that if the macro in Listing 4.2 is used for a function then it is not possible to use the macro from Listing 4.1 for that same function as this would lead to a double definition of access control.

4.2 Performance

In this section, we will display and compare the access control module's performance and compare the result to the second requirement set up in Section 1.2.1. The data was collected by running Satcube's program as normal, with the addition of the access control module, and measuring the execution time. The same program was also executed with a cache enabled for additional comparisons. The cache was introduced so that the application could save previous query requests and was implemented in the following way:

Function	Median	Average
f_read	266 ms	295 ms
f_open	258 ms	311 ms
f_write	248 ms	352 ms
f_close	255 ms	292 ms
xTaskCreate	254 ms	308 ms
vTaskDelete	235 ms	292 ms
xSemaphoreCreateCounting	292 ms	293 ms
xQueueCreate	281 ms	288 ms

Table 4.2: The average and median execution time of access requests for some of the listed functions

Whenever a user's access request is evaluated, the result, the user, and the action are stored in a list each. Before consulting DATALOG, the monitor checks the logged lists if the user has previously requested access to the same action. If the user and corresponding action have been logged, the stored result is returned. However, if no logged request is present, DATALOG will run the query to produce a new result which will be stored in the cache. The cache is limited in length and the entries will therefore be subsequently replaced.

Requirement 2 states that each query should take no more than 100 ms to compute and from the collected data we can conclude that the solution does not uphold this requirement. From Table 4.2 we can see that each access control query has an average execution time of between 352 and 288, which is 188 ms more than the requirement at its lowest. This long execution time is most likely the result of slow execution in the DATALOG part of the program. To test this hypothesis we implemented a simplified table. While this implementation lied outside the scope of the thesis we argue for the necessity to accurately resolve the cause for the poor performance. The permission table was implemented using two arrays in regular C-code, the first array held the list of users while the second held the permissions. Thus to evaluate an access request, the first list is searched for the appropriate user and then the second list for the permission of that index. If a match is found, access is granted and is refused otherwise. In DATALOG each post in the table corresponds to a fact written in DATALOG in the form: *authorize(user, action)*. While the table written in pure C-code was close to instantaneous (0-1 ms), the DATALOG-table still suffered from relatively poor execution time (20-40 ms). We, therefore, argue that the chosen DATALOG-engine impairs the performance of the access control module.

Using the caching technique, we produce a simplified table based on RBAC. RBAC with DATALOG form the evaluation process of new queries and then the result is stored in a list associated with the appropriate pair of users and actions. Furthermore, this version also allows for negative authorizations such that if a process has previously been denied access to an action, it will be further denied if the attempt is still logged. The performance of the cached solution is presented in Table 4.3 and it's evident that the cache decreases execution time to an extent that the module now fulfills the requirement of execution time below 100 ms.

Listing 4.3 shows the program which was used to evaluate permissions on the device

Function	Max	Average
f_read	311 ms	20 ms
f_open	340 ms	30 ms
f_write	262 ms	49 ms
f_close	371 ms	25 ms
xTaskCreate	195 ms	17 ms
vTaskDelete	298 ms	36 ms
xSemaphoreCreateCounting	344 ms	38 ms
xQueueCreate	346 ms	28 ms

Table 4.3: The maximum and average execution time of access requests for some of the listed functions with cache enabled

during benchmarking. Among other things, Satcube's device hosts a web-server which is controlled by the the tasks *Web Main* and multiple worker tasks *Web WT*. From the web server, most of the device's files, such as settings, can be accessed which works as intended since *Web WT* is member of *Fsread* granting the permission to read files. However, if an attempt to change the settings file were made, the action would be denied as *Web WT* is not a member of *Fswrite*. This is verified by checking the log which register every failed attempt to access a protected action.

```
1  role("Root").
2  role("Fsread").
3  role("Fswrite").
4  role("Kernel").
5
6  user("Web␣WT").
7  user("Ethernet").
8  user("System␣Control").
9  user("MbedWeb␣Main").
10
11 action("f_open").
12 action("f_write").
13 action("f_close").
14 action("f_read").
15 action("xTaskCreate").
16 action("vTaskDelete").
17 action("xSemaphoreCreateCounting").
18 action("xQueueCreate").
19
20 member("Ethernet","Root").
21 member("Web␣WT","Task").
22 member("Web␣WT","Fsread").
23 member("Web␣Main","Kernel").
24
25 privilege("Task","xTaskCreate").
26 privilege("Task","vTaskDelete").
27 privilege("Kernel","xSemaphoreCreateCounting").
28 privilege("Kernel","xQueueCreate").
29 privilege("Fsread","f_read").
30 privilege("Fswrite","f_write").
31
```

```
32 privilege("Root", X) :- action(X).
33
34 directInherit("Kernel","Task").
35 directInherit("Fswrite","Fsread").
36
37 inherit(A,B) :- directInherit(A,B).
38 inherit(A,B) :- directInherit(A,C), inherit(C,B).
39 member(A,C) :- inherit(B,C), member(A,B).
40
41 authorized(A,B) :- member(A,C), privilege(C,B).
```

Listing 4.3: The DATALOG program that was used in testing of the module

4.3 Physical user

In this thesis, the focus of access has been mostly on the individual process in a FreeRTOS system as this best represents the environment of the case-study at Satcube. However, one can still use the access-control module with a policy that is modeled for physical users instead. It is possible to do this in two different ways in the current state of the solution. The first way is to represent different users with different DATALOG-programs such that when a user logs in to the device, the program that is used is to evaluate the user's permissions is swapped to match the current user.

The second way is to supply the access control task with the name of the user who is logging in, thus the access control's task-name reflects the user and can be used in the DATALOG-program to model permissions. Physical users' relationships can often be ordered in a hierarchical structure of roles making it easy to order privileges based on the role which makes it natural to use the hierarchical RBAC-model for physical users. An example of one such program can be seen in Listing 4.4, where tasks are initialized with the names: *Alice*, *Bob*, and *Carl*. This program allows the user *Alice* to do a set of file system actions, *Bob* to do everything Alice is allowed to do and the protected action *start_transmission*. Finally, *Carl* is allowed to do *factory_reset* and every other function because he inherits memberships from both *User* and *Admin*.

```
1  role("User").
2  role("Admin").
3  role("Factory").
4
5  user("Alice").
6  user("Bob").
7  user("Carl").
8
9  action("f_mkdir").
10 action("f_write").
11 action("f_open").
12 action("f_close").
13 action("f_read").
14 action("start_transmission").
15 action("factory_reset").
16
17 member("Alice","User").
```

```
18 member("Bob","Admin").
19 member("Carl","Factory").
20
21 privilege("User","f_write").
22 privilege("User","f_open").
23 privilege("User","f_close").
24 privilege("User","f_read").
25 privilege("Admin","start_transmission").
26 privilege("Factory","factory_reset").
27
28 directInherit("Factory","Admin").
29 directInherit("Admin","User").
30
31 inherit(A,B) :- directInherit(A,B).
32 inherit(A,B) :- directInherit(A,C), inherit(C,B).
33 member(A,C) :- inherit(B,C), member(A,B).
34
35 authorized(A,B) :- member(A,C), privilege(C,B).
```

Listing 4.4: An example program using actors' names as task names

5
Discussion

The goal of this project was to develop an access control module with a set of requirements to guide us. The module prevented every illegal access request made during the tests in Section 4. As such, the module succeeds in fulfilling the fundamental requirement from Section 1.2.1. In the following chapter, we discuss the answers to the questions from Section 1.2 and how well we managed to fulfill the rest of the requirements based on the presented results from Section 4.

Semantics.
It is clear from the results that it is possible to implement access control without interfering too much in an existing codebase. In the case study at Satcube, the access control module could be implemented without any tailoring of the code to fit. The module was initialized, and included where the access control was needed and then functioned as intended.

Performance.
From Section 4.2 we can conclude that the access control module operates properly when it comes to controlling access to certain functions but fails to fulfill the performance criteria. One can argue that the requirement is upheld if we allow queries to be stored in a cache, however it should be recognized that this is an unsafe way to handle access rights in any system. Although the cache was implemented and tested, it was made to test the cache as a concept and to help the testing of the module under the pretense of evaluating the queries on a faster logic engine. Caching access requests is generally an unsafe idea as the circumstances surrounding the code may change and make the subsequent request unlike the first, it is, therefore, a security risk of allowing a process to have the same access as it had on the previous call.

Is RBAC a suitable policy to use when introducing access control in an embedded environment?
It is our opinion that RBAC is a suitable policy model when modeling access control for an embedded system. Functions often belong to some category by which they can be labeled and therefore divided. For example, given a set of functions handling the file system we can have a role for reading files and another for writing files. Furthermore, in the kernel, we can divide functions as task handling, synchronization, and message queues. These roles can all in turn be inherited by a super role: *Kernel*. Each process or task in an application can similarly be labeled users that can be assigned to the roles.

Can access control be done efficiently in an embedded system using the specified language and mechanics?
The results gathered from running the module at Satcube shows that using Datalog with RBAC was not an efficient solution. Even though we were granted a generous execution

time limit of 100 ms, this expectation could not be met. It can be argued that given a better logic engine for DATALOG, an efficient solution could be possible. However, as no other logic engine was implemented, no conclusions can be made. RBAC on the other hand is a mature policy model that allows for policies that can be small in size but still allow for a complex permission and user-layout.

Can access control be easily introduced in an existing codebase?
Macros allow us to easily introduce access checks with the inclusion of a header file into a preexisting codebase. This can be done without changing the semantics of the code. However, as has been pointed out in Section 4.2, the module introduced additional overhead on each protected system call. Also considering that some system calls will unavoidably be called in subsequent order e.g *f_read* will be called after *f_open* after which *f_close* will be called to close the file resulting in an execution time of one second just to get a file's content. If the target application is time-sensitive and the evaluation of access requests is slow, it could cause serious degradation or even failure to execute the code to the extent of the system shutting down. We argue that introducing access control into applications that are written without the consideration for such control proves difficult and should be done with great care, but still emphasize that it is entirely possible.

5.1 Related work

Binder is a concept of a security language which uses DATALOG to construct certificates in distributed systems where multiple databases may exist and access needs to be maintained over several instances. The concept is argued to be a more readable way to form policies and certificates thanks to the use of DATALOG which has polynomial execution time and simple semantics allowing for policies to be easily read and well-defined. [4]

Microsoft has recently begun support for access control to be handled in an RBAC policy model in its cloud-based platform Azure. The service is called Azure RBAC and provides fine-grained control over the resource manager. The service has several predetermined roles which helps determines who can access and what they can access.[16] Similarly Conjur offer solutions for access control to applications, tools, critical infrastructure and other sensitive data by using RBAC as its model. [3] Much like Azure, this is a cloud based service and therefore the requests are handled off-device. These solutions, therefore, are better suited for when physical users requests access to a non-local resource but can not be used as efficiently in access control for local processes.

This cloud-computing issue has been recognized in [23] where the authors propose a new access control strategy where the evaluation of policy is done in the edge computing layer. The edge computing layer is explained as a set of on-site devices that do the necessary computation for the access evaluation. This strategy is primarily directed towards the IoT and to decrease the effect that the latency may have on performance. However, this strategy has the same drawback where a device may need a strictly local access control mechanism.

Much research has been poured into how to expand the expressiveness of RBAC. One such study introduces a new model: Priority-Attribute-Based RBAC (PARBAC for short) which adds priority attributes to the existing role-based model. The authors of the study claim that by attaching each role with priorities and attributes it lowers the operational

time and is ideal for large organizations. [22]

In [14] the authors implement RBAC in an embedded system using the Linux Security Module. The endeavor succeeds in a module that allows for access control with an overhead of 0.8 ms. This fast result is achieved by compiling the RBAC file into hash lists similar to our implementation of the simple permission table presented in Section 4.2. Given a large set of users and actions, this solution requires large amounts of space in the system. Furthermore, the policy cannot be modified after compilation, and is susceptible to birthday attacks.

5.2 Conclusion

To conclude, the developed access control module based on RBAC and Datalog could be easily integrated with a preexisting embedded system with FreeRTOS as its operating system. However, the performance degradation attributed to the Datalog logic engine makes the final product an unattractive option for embedded systems unless a faster evaluation engine is used. Even though fixes to this performance issue in the form of caching access requests were made and had promising results, it may not be ideal to circumvent the evaluation of the policy in the sake for performance.

RBAC as a policy model, is an attractive choice for embedded systems as functions and tasks are often easy to divide into roles and responsibility. However, as the amount of actors, roles and resources may grow when expanding a system to include the ever-changing climate of IoT, another model may be required.

Datalog is a mature language with multiple implementations across several languages such as Java, Rust, and Python. Its semantics makes it easy to use and understand since it is based on formal logic and makes policies well defined. Datalog as a policy evaluator requires a well-performing logic engine to prevent introducing performance issues and causing bottle-necks in the execution.

5.3 Future work

The most probable cause of the poor performance presented in Section 4.2 is the Lua-engine used to implement Datalog. A natural future step to improve access control using RBAC and Datalog is to either develop or port an existing Datalog engine that boosts performance such that queries are no longer bottle-necking the execution with long evaluation times. Furthermore, expanding on the idea of a newly developed Datalog-engine, it could be beneficial to use a variation of Datalog such that it enables the use of negation and constraints as in [13, 15]. Using constraints in Datalog would allow the definitions of file structures to implement a more fine-grained control of access to files.

Lastly, while RBAC is a widely used model and allows for a more compact declaration of permissions and privileges, much research has been invested into finding a more developed model that fits the ever-changing relations in an IoT-model. These investments have resulted in several versions that has the potential to be the next established access control model [10, 22]. It would be interesting to see if an evolved version of RBAC could find

use in an embedded system whose rights would need to encompass more and more users in an IoT-model.

Bibliography

[1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[2] Moritz Y Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.*, pages 159–168. IEEE, 2004.

[3] Conjur. `https://www.conjur.org/`. Accessed: 2020-09-14.

[4] John DeTreville. Binder, a logic-based security language. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 105–113. IEEE, 2002.

[5] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[6] FreeRTOS. What is FreeRTOS? `https://www.freertos.org/about-RTOS.html`. Accessed: 2020-05-25.

[7] Sean Gallagher. How one rent-a-botnet army of cameras, DVRs caused internet chaos. *Ars Technica*, October 2016.

[8] Joseph Y Halpern and Vicky Weissman. Using first-order logic to reason about policies. *ACM Transactions on Information and System Security (TISSEC)*, 11(4):1–41, 2008.

[9] Vincent C Hu, David Ferraiolo, and D Richard Kuhn. *Assessment of access control systems*. US Department of Commerce, National Institute of Standards and Technology, 2006.

[10] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, 2005.

[11] Brian Krebson. The democratization of censorship. *Krebs on Security*, September 2016.

[12] Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ansi standard on role-based access control. *IEEE Security & Privacy*, 5(6):41–49, 2007.

[13] Ninghui Li and John C Mitchell. Datalog with constraints: A foundation for trust

management languages. In *International Symposium on Practical Aspects of Declarative Languages*, pages 58–73. Springer, 2003.

[14] Jae-Deok Lim, Sung-Kyong Un, Jeong-Nyeo Kim, and ChoelHoon Lee. Implementation of lsm-based rbac module for embedded system. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *Information Security Applications*, pages 91–101, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[15] Xiaofan Liu, Natasha Alechina, and Brian Logan. Expressing user access authorization exceptions in conventional role-based access control. In *International Conference on Information Security Practice and Experience*, pages 233–247. Springer, 2013.

[16] Microsoft azure rbac. `https://docs.microsoft.com/en-us/azure/role-based-access-control/`. Accessed: 2020-09-14.

[17] NIST. Role based access control. `https://csrc.nist.gov/projects/role-based-access-control`, November 2016.

[18] Matunda Nyanchama and Sylvia L Osborn. Access rights administration in role-based security systems. In *DBSec*, pages 37–56. Citeseer, 1994.

[19] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[20] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[21] Hui Suo, Jiafu Wan, Caifeng Zou, and Jianqi Liu. Security in the internet of things: a review. In *2012 international conference on computer science and electronics engineering*, volume 3, pages 648–651. IEEE, 2012.

[22] A. Thakare, E. Lee, A. Kumar, V. B. Nikam, and Y. Kim. Parbac: Priority-attribute-based rbac model for azure iot cloud. *IEEE Internet of Things Journal*, 7(4):2890–2900, 2020.

[23] Ronghua Xu, Yu Chen, Erik Blasch, and Genshe Chen. A federated capability-based access control mechanism for internet of things (iots). In *Sensors and Systems for Space Applications XI*, volume 10641, page 106410U. International Society for Optics and Photonics, 2018.