MASTER'S THESIS 2020

Secure Updating of Configurations in a System of Devices

Jens Mellberg

ISSN 1650-2884
LU-CS-EX: 2020-04

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-04

Secure Updating of Configurations in a System of Devices

Jens Mellberg

Secure Updating of Configurations in a System of Devices

Jens Mellberg
dat14jme@student.lu.se

February 10, 2020

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Mattias Nordahl, mattias.nordahl@cs.lth.se

Examiner: Boris Magnusson, Boris.Magnusson@cs.lth.se

Abstract

Large scale systems of interconnected IoT devices are widely used for performing important tasks, all from data collection and analysis to performing work tasks in certain fields. Such systems are often subject to frequent updates, due to e.g. changes in functionality, bug fixes or security issues. Releasing a new update means having to update many individual devices, that might not be available at the same time, while minimizing any loss of data or system downtime. Furthermore, updates need to be verified as trusted in order not to compromise the integrity of the system. Additionally, as software is deployed in a system of devices, issues such as incompatibility between devices, and other information regarding the combined software of the involved devices may not be identifiable until after initial deployment. Therefore, it would be practical to use a format that allows for documenting these issues together with the software after its release without changing the software itself or compromising the verification process. This thesis presents a design for ensuring the integrity of software components taking part in updating a system of devices. A format that allows for continuous addition of meta information to a software after its release is also presented. The results are then evaluated against a real life scenario.

Keywords: IoT, PalCom, automatic update, update verification

Acknowledgements

Special thanks to Boris Magnusson and Mattias Nordahl for supervising the project and providing valuable feedback when writing this thesis as well as helping me set up and understand the PalCom framework. This work was done as part of the SSF-project RIT17-0035 Smarty.

Contents

Chapter 1
Introduction

Digitalization has greatly simplified many aspects of our society by using software to perform tasks for us. It has also given rise to many systems of computer devices, continuously running software, all while communicating with other devices and servers, often referred to as "always on" systems. A big share of these are connected wirelessly which adds additional challenges because of devices not always being available due to connection issues, being in use or turned off, amongst other things. One of the major issues is updating these devices. In particular a system of devices with dependencies amongst themselves might have to be updated simultaneously, but some devices might be unavailable at the time. Nevertheless we want the system to update smoothly without downtime. This means that the system must be able to fully update every aspect of itself while still maintaining full functionality. Additionally, to support scaling of the system structure, there is a need to be able to fetch updates from a close source and not rely solely on a central database. At the same time this also has to be done in a secure way, since there is always a risk of attackers trying to inject malicious data posing as an update. This is even more relevant on a wireless system due to all data being transmitted through the air, giving outsiders an easier access to infiltrate the system.

Another issue with using software in a large scale system is that of compatibility with other components. After a new software version is released, other users may discover that it is not compatible with certain architectures or other running software. Additionally, we would like to allow users to "sign" the software when it has been tested in their particular setting. We therefore want to be able to add these kind of annotations to the affected software version incrementally without changing the software itself.

As a way of simplifying developing applications for IoT systems by abstracting certain aspects, middleware are often used. Middleware provides additional services

to software and connects devices together and are often used as underlying software. One kind of middleware aiming to ease the updating process is the PalCom architecture model, developed at Lund University. Our design will be integrated on the PalCom architecture. PalCom includes a configuration server that will release new configurations to its clients which they should update to. A PalCom configuration is a set of software (referred to as services) that the client is instructed to run. These software will be fetched from the server, or for efficiency, from other clients as well. They must therefore be verified to ensure that they have not been tampered with and are correct.

The security of a software system is arguably one of its most important aspects, since an intrusion could compromise the system as a whole and even shut it down. This could have devastating effects if the system performs an important task like for example keeping hospital records. The system might also include sensitive personal information that might be leaked in case of a breach. Software security is also a field that needs constant research and improvement since the opposing side, malicious attackers, is constantly coming up with new methods.

Finding a secure and efficient way of implementing and evaluating these security aspects will be the primary focus of this thesis.

Chapter 2
The updating process: Previous work

The problem of updating software has been around since computers started making their way into mainstream use. To better illustrate the problem, we will look into the challenges posed and see how they have been solved previously in various settings and how one can make sure that the new software is trustworthy.

In the days before the internet distributing and updating software was done manually by disk or CD-Rom. Security was less of an issue then since the steps from acquiring the updates to installing them was carried out by humans and the process was much less automated. With the introduction of the internet however, updating software became more convenient while also introducing new challenges. New versions of applications are now available online for easy access but this information is now sent over a public network (the internet) and is vulnerable to external modification. In the current day, software updates are often completely automated. An example is smartphone applications which can be set to automatically update themselves as long as certain conditions are met. Package Managers[1] can also be used, which deals with software in packages. These contain information such as the software's dependencies and allows for automatic installing and updating. This opens up even more vulnerabilities as the user are completely detached from the process and can not aid in detecting anomalies.

A common way of automating the update process is to have the application regularly request updates from an update server, where the DNS name of the server is hard-coded into the application beforehand[3]. The update is then fetched and installed, typically requiring the software to do a complete restart. This is usually acceptable for most applications used by the average user. To prevent other attacks this process also requires extra measures taken such as authenticating the server (DNS responses can be spoofed) and validating the software received.

A solution to one of the more complex scenarios of the updating problem is posed by researchers at the University of Erlangen–Nürnberg, where they explain and attempt to give a solution to dynamically updating the software of embedded systems[2]. Devices in these kinds of systems may be under heavy resource constraints as well as having software running that is constantly doing important tasks. The system could be designed to run for a very long time and regular updates to the software might be required to fix issues or adapt to changing circumstances. Doing a complete restart of the software when updating is usually undesirable in these scenarios since it would lead to either losing the current application state or having to spend resources to saving and restoring it. Their solution is to gradually update the code by identifying the changes in the updated code and analysing the current execution of the program to see how the changes will affect the flow. This involves advanced analysis of the device's memory such as checking if the current running function is affected by the update or if any functions present on the call stack will have their return addresses altered.

This problem becomes even more complex when we are working with big interconnected IoT systems that are constantly communicating data between the devices in the system. If a new update is released we would ideally want to instantly update every device to the new configuration without any downtime since it could result in loss of gathered data. But that is not realistic. Each device will finish their individual update process at different times and some devices might even be offline. To illustrate the problem, consider the following example:

An IoT system consists of a central server and several client devices that collect data. The clients send their data unencrypted to the server for analysis and logging. The system is later updated to include encryption for heightened security. This will lead to one of the following problem scenarios:

1. If the server is updated before the clients it would start to expect encrypted data and treat each message as encrypted, but the devices are still running the old configuration, and will send the data unencrypted.

2. If the clients are updated before the server they will start sending encrypted data while the server will expect normal messages.

Both cases will lead to the server being unable to process the data and it would be lost, or cause the system to behave unexpectedly. Wißbach et al. (2016) provides their solution to this, which involves each device having their own Local Update Manager (LUM)[7]. New updates are first sent to the LUMs of all affected components of the system. These will communicate with each other and wait until all affected LUMs have correctly received their update. The LUMs then instructs the components to stop communication with the affected devices and starts the updating process on them synchronously. The LUMs will keep communicating with each other and when all updates have been completed successfully, the system will resume its active state.

Their solution does however require that all affected devices are eventually online

simultaneously, which might not be possible in an environment where devices are often disconnected from the system. Therefore PalCom provides an alternative solution for the problem which is to update the systems components asynchronously, while letting the server run the old and the updated configuration simultaneously. This removes the coordination needed to assure that each device is able to complete the update at the same time, and any unavailable or busy device can simply delay the update while still maintaining a fully functional system. The drawback is that the system components have to be designed in such a way that both versions can be handled simultaneously. This might be impossible in certain cases, for example if a resource cannot be accessed by two processes simultaneously. Although it appears that such a design is possible in most systems.

Chapter 3
Problem Formulation

We will use a running example throughout the thesis, based on a real life scenario from e-health.

ItACiH[6] is a project that focuses on it-support for healthcare when patients are treated in their homes. Traveling nurses use mobile tablets with a software application that contains their visiting schedule, patient information, test results etc. They will perform check ups and medical tests on the patients and can register and view the results on their tablets. All results are sent to the hospital where they can be analysed by doctors. This system is built on top of the PalCom framework and both the patient's and nurses' tablets are connected to a central configuration server. Note that this is a simplified example, real life applications could include additional servers. Lets imagine a scenario where the software used in the nurses' tablets has recently been patched to fix a bug and this update should now be distributed to all the affected tablets. When these updates are received by the tablets, they are installed and begins executing. The communication process must be designed in such a way that it allows the server to communicate with tablets running both the old and new configuration. The tablets need to verify that the software they received is indeed the correct one that the configuration server has assigned them. The administrator of this system then discovers a compatibility issue between the software and the operating system that the tablets are running that causes the software to crash unexpectedly. He then needs a way to attach this information about this issue to the software without changing it in a way that will disrupt the verification process. In addition to compatibility, it is useful to have the option for the administrator of approving that the software is now tested in this particular system. These attributes are associated with the software, but are not necessarily fit to be part of the software itself. Therefore we want a representation that allows us to incrementally add information to the software even after its release. To summarize, this thesis will attempt to solve the following problems:

- Each device needs to verify that the software they received is authentic and of the correct version.
- We need a way of attaching external information to each software without changing the software itself or breaking the authenticity validation.

Approach

We will start by looking closer at the PalCom architecture to see how the updating process is performed in more detail and look for potential vulnerabilities to determine at which step a validation check should be performed. Then we will provide a general overview of cybersecurity threats to get an idea of what we need to consider before implementing the update verification in the architecture. This includes researching common exploits and attacks as well as existing verification methods to ensure that the correct data gets delivered. Later we will implement this verification as well as the functionality for adding additional data to the software incrementally after its release. Finally we will evaluate the results and provide a conclusion.

Chapter 4
PalCom

To continue further we need to know more about how the PalCom architecture is structured. The system is far too big to describe it in detail and therefore we will focus only on the parts that are relevant for the thesis.

General Overview

PalCom acts as a middleware for simplifying connecting devices together in a system and handles the low level communication between devices as well as updating and instantiation of the software that they are running (this software is referred to as services by PalCom). We will not go into detail about how the connections between devices are established and what communication protocols are used and instead focus on how services are deployed and updated. This process is built around a central configuration server and client devices that are connected together in a system. The configuration server keeps a table with information about which configuration each of the system's devices should currently be running. Note that, in the context of the PalCom framework, we define a configuration as a set of services of specific versions that a device should run. This server will act as the coordinator of the system and will communicate to each device which configuration to run. To update the system, new configurations can be assigned to the client devices. The server will then communicate this to all affected devices and they will in turn download needed services as specified by the configuration and start running the new versions.

When a device does not already have a service in the particular version that is included in its configuration, it should be able to download it from other available devices. A device should potentially accept and store any service that is sent to it for future use. This means that we do not know if the currently stored services contain corrupt data or malicious code. Thus some sort of verification becomes necessary

to ensure that the services can be trusted and are safe before they are executed.

The PalCom framework includes tools for visualizing, administrating and debug-

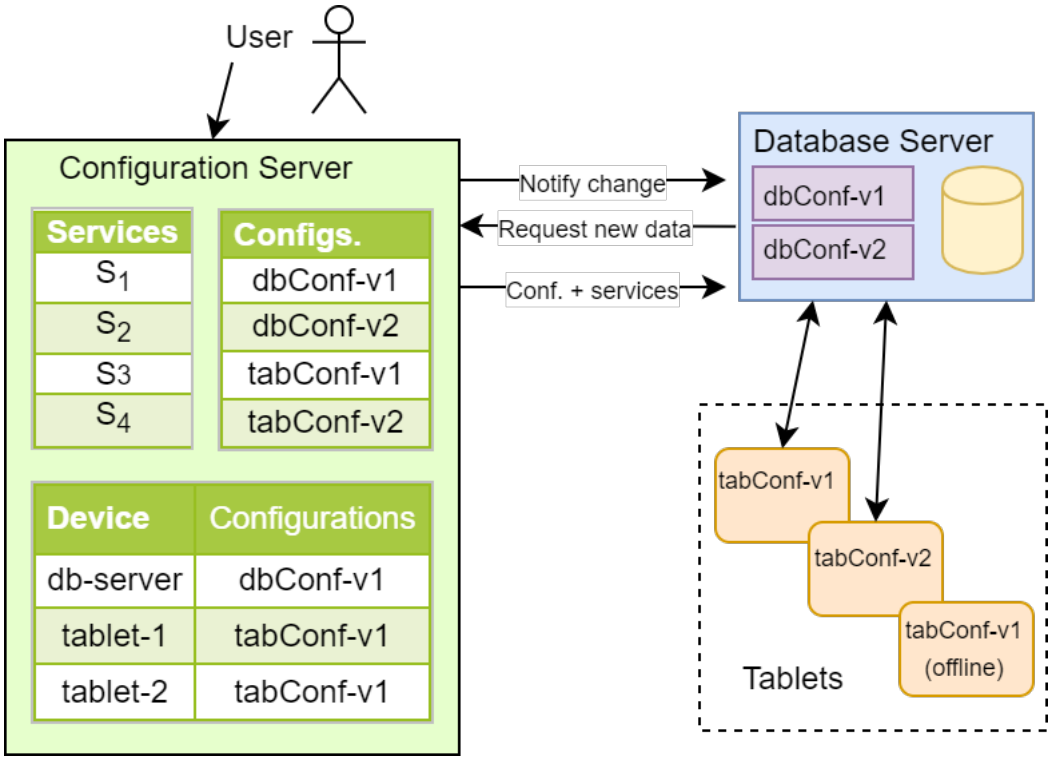

Figure 4.1: The configuration server keeps a table of which configuration each device should be running. Whenever a change in the table is made, the configuration server notifies the devices about this and provides the configurations.

ging purposes. One of these is the PalCom Browser which lets you explore all currently active devices on your PalCom network and what services they are running. PalCom services are self-descriptive. They contain an API that the PalCom browser can visualize in a graphical interface. Users can interact directly with the system services to try out and learn how they function. We can also simulate devices with another PalCom tool called "TheThing". This tool will start a virtual device that will act in the same manner as a physical device. We will use both of these tools for making sure services and configurations are received correctly later.

Technical Details

The PalCom framework is written in Java and uses its own format for serializing and unserializing java classes such that they can be sent between devices, called PON (PalCom Object Notation)[5]. The framework provides tools to convert every PalCom class to and from this format. This is how a configuration (and all other PalCom objects) will be sent to a device from the configuration server. The device will store the configuration in its local file system in the PON-format so that it can

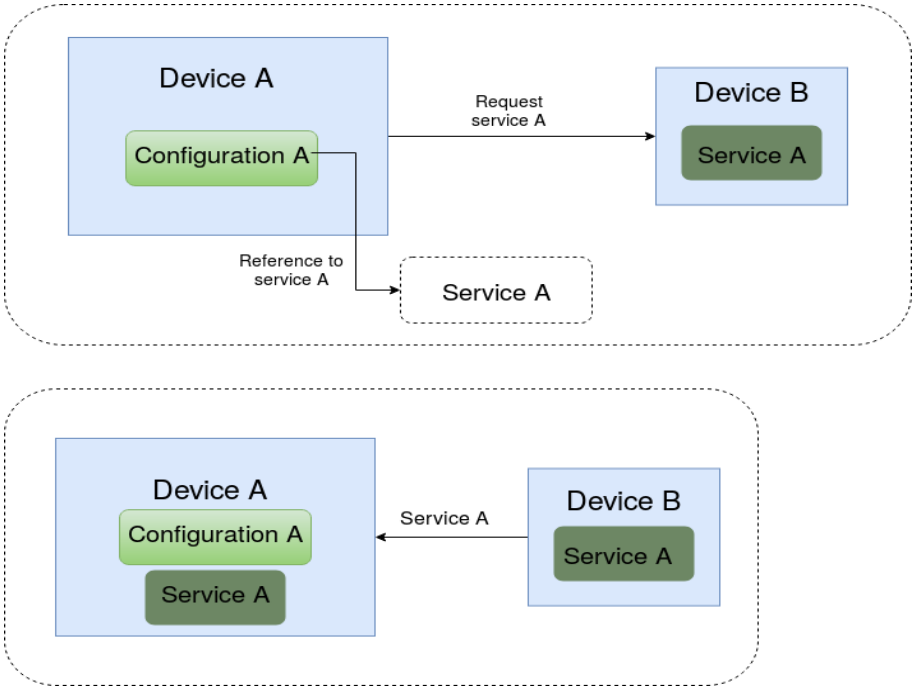

Figure 4.2: Device A has received a configuration. This configuration contains a reference to Service A (note that it is a reference and not the actual service). Device B then provides this service to Device A. Device B is an untrusted source and the service needs to be authenticated.

be loaded each time it boots up. The services are currently implemented as jar files and their content is not restricted so that users of the system are free to develop suitable services to fit the purpose of the system.

The term "service" can mean and refer to several things depending on the context in which it is used, for example:

- A general type of software. Devices can host and run `services`
- The implementation of a specific service, i.e a jar-file.
- An instance of a running service.

To help dealing with this ambiguity and to be able to be precise when referring to things, the PalCom framework uses the following hierarchy of terminology:

- `Kind`
  This refers to what kind of resource something is, e.g. a device, service, configuration etc. This is the most fundamental distinction and all latter terms will be built on top of the previous ones.
- `Type`
  A kind of resource, such as a service, is of a certain type. PingService and CameraService are both services, but of different `types`.

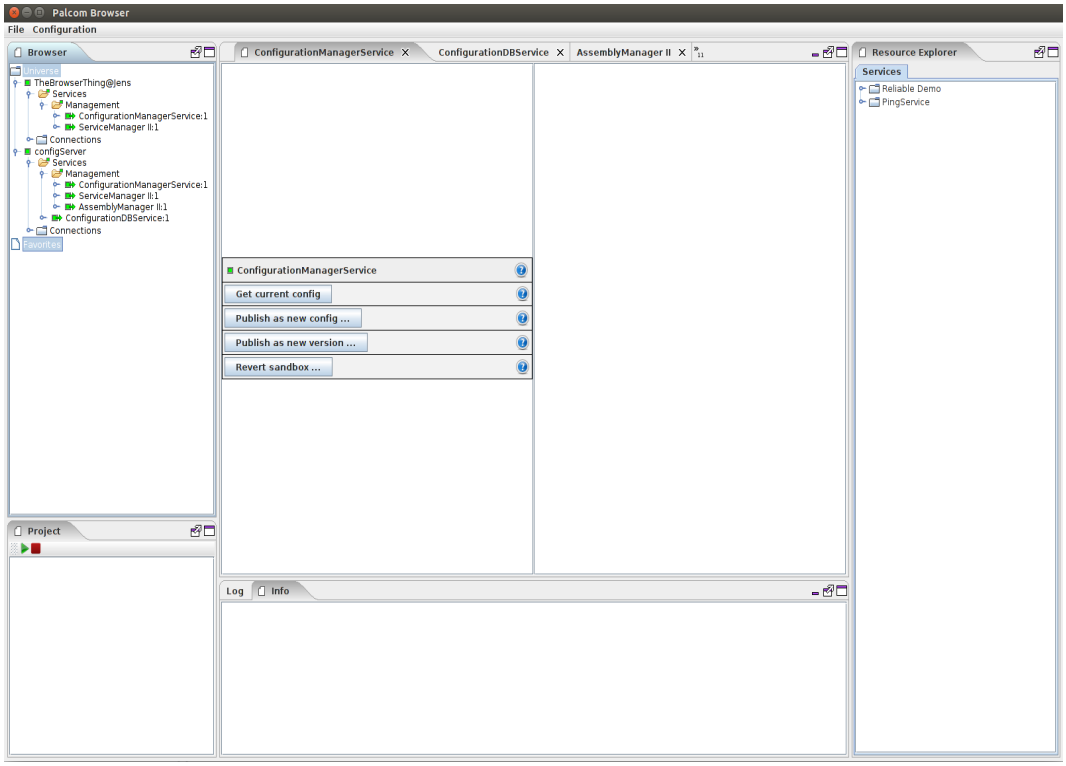

Figure 4.3: The GUI of the PalCom Browser tool. The browser provides a visual interface for interacting with services and configurations

- `Version`
  A type can appear in different versions.

- `Instance/Role`
  A specific version of a given type can exist in multiple instances. We give them roles (or instance names) to distinguish them.

Additionally there can be references to a resource, which is an object that "points" to a specific resource and is separate from the resource itself. These are used by configurations to point to their resources. Note that this is not a hierarchy in the sense that they are Java classes inheriting each other but expresses aggregation. To better understand the hierarchy, we provide these examples. A `ServiceTypeVersion` is the actual implementation of a service. It is of a specific kind (service), type (e.g camera) and version (1.0). In practice it is the service jar file. We can reference it with a `ServiceTypeVersionReference`. Similarly, we can reference a configuration implementation with a `DeviceConfigTypeVersionReference`. To generalize these for all kinds, we use the terms `TypeVersion` and `TypeVersionReference`.

A service implementation will also contain a reference to itself. It is used for comparisons with references contained in configurations to determine if a matching service (`ServiceTypeVersion`) exists on the device. An update interaction will begin with

the configuration server notifying a device that its configuration has been changed or that a new one has been added. Alternatively a device will request this information when it first comes into contact with the server. The configuration server will then inform the device of which configurations it has been assigned according to the servers internal table. These are referred to as bindings and a device is bound to certain configurations. The client will then store these as pending while it gathers the required resources. The bindings include resource references to the configurations which in turn include references to the services. The device will use the information from each reference to construct a unique file path for where the resource should be located. If a file for each specific file path exist, the device will conclude that it has all the required resources, otherwise it will request any missing resource. When all required resources has been acquired, the configuration will switch from pending to active and all services will be loaded. Somewhere during this update process each acquired resource needs to be verified.

Summary

To summarize, the PalCom architecture consists of a Configuration server and several connected clients. To inform the clients of which software to run the server will send configurations to each client. A configuration contains references to specific software versions (services), which the client can use to request the actual service implementations. Since we cannot trust that a source actually provides the service pointed to by the reference, we need some way of validating its authenticity to avoid executing potential malware.

Chapter 5
Cybersecurity threats and prevention

Before implementing the validation in the system it is necessary to have some background knowledge about cybersecurity, about which exploits exist and how to prevent them.

Security attacks are becoming more and more sophisticated, meaning that security itself has to keep evolving to protect against these new threats. This, in turn, forces attackers to invent newer exploits again, essentially creating a never ending cycle of keeping both fields in need of constant improvement. This makes computer security a fairly unique field in the sense that it is more dependent on how the security exploits are developing rather than simply advancing the field by learning more about it.

Given this relationship, it is impossible to prove that a system is completely secure. The closest we can come is to argue how the system is safe against the attacks that are known. Therefore this section will include a description of common attacks and establish which of them that PalCom might be vulnerable to.

Man-in-the-middle Attack

The Man-in-the-middle Attack (MITM) is an attack where a third party will insert themselves in between the two communicating devices. It will then receive the messages sent from either of the devices, read them or possibly change their contents and then pass it on to the original receiver. All while the two devices believe they are communicating directly with each other. This can be achieved if the message when the parties exchange keys are able to be intercepted. The attacker might acquire the correct encryption key from party A and then proceed to send its own key to party B. Party A will then believe that this is B's key and send messages that are

encrypted with it. The attacker can then decrypt these, manipulate them, and then send a response to B encrypted with their actual key.

A way to defend against this attack is to somehow have predetermined keys that never are exchanged over an insecure channel.

Malware

Malware is a widely known security threat. This is software designed to hurt the system it is running on and can come in many forms. Protecting against this threat is not always straight-forward and vulnerable points in the system need to be analysed, such as carefully analysing when software that has been received from other sources is executed.

Checksums and Hash functions

Checksums are a way of detecting errors in a chunk of data, by producing a sequence of control bits from the data. If this sequence matches the expected one, it can be assumed that the data is correct. One of the common ways of achieving this is by using Hash functions. Hash functions produce a seemingly random string (called a hash) from an input by performing a sequence of mathematical operations on it. It is critical that the original input data can not be produced from the hash (or at least that it takes an unreasonable amount of time to do so). A well designed hash function should also have few collisions, meaning that different data sets should be very unlikely to produce the same hash. If collisions were to be able to be found, a malicious piece of software could be made, which produces the same hash as the original program. The error detection will then fail to detect anything and the program will be assumed to be correct. There are three ways of looking at the security of hash functions, these are:

- Pre-image resistance
  Given the hash, it should be hard to find any data which produces that hash.

- Second pre-image resistance
  Given an input, it should be hard to find a different input which produces the same hash.

- Collision resistance
  It should be hard to find any two different inputs which produces the same hash. Figure 5.1 shows the probability of finding collisions in the hash function Sha-256 for the number of hashes performed. As we can see, closer to $6 \cdot 10^{44}$ hashes needs to be performed to guarantee finding a collision, which will take an extremely long time to generate. This makes Sha-256 a reliable hash function when used for hashing input values with high entropy inputs (and will be for many more years unless an internal flaw is discovered[10]). A side note

is that having a hash function that performs well at these resistances does not make it a fitting candidate for all purposes. For example, using Sha-256 for password hashing is not recommended since it takes a short amount of time to calculate and passwords generally have a fairly low entropy. This makes it weak against attacks such as dictionary attacks[11].

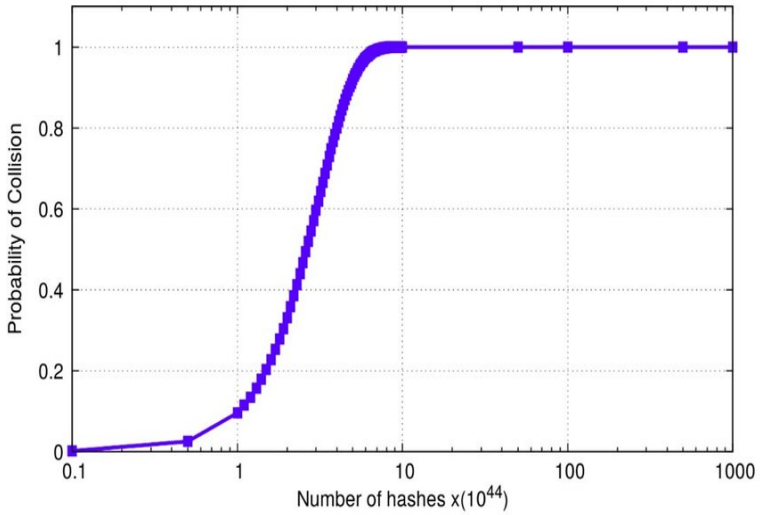

Figure 5.1: Sha-256 collision resistance. Probability of a collision given a number of hashes.

Avoiding collisions completely is not reasonable as that would require the total amount of output hashes to be the same as the amount of possible inputs, which is highly impractical. Instead the security is based on having the collisions be hard enough to find that it will be infeasible using the current technology. The frequently used size of the output ranges from 128 to 512 bits. Figure 5.2 shows a collision found with the Sha-1 hash function. The two pdf documents produces the same hash and was discovered by researchers at Google[9]. This requires developers to keep up with any recent news about flaws found in hash functions.

Digital Signatures

A digital signature is similar to a hash function in that it produces a sequence of bits from the data, but with the addition of asymmetric encryption. One party will sign a chunk of data using an algorithm that produces an output from the data and their private key. This signature is then sent alongside the data and any receiver can use the corresponding public key to verify that the signature belongs to the data. As opposed to using hashing, digital signatures guarantee that the message was sent from the correct source, since creating the signature requires knowledge of the private key.

Figure 5.2: Two different Pdf documents producing the same Sha-1 hash.

Summary

Data sent over any network is always susceptible to being compromised. In the case where it is crucial that the data has not been changed since the message was sent, a proper key distribution protocol should be used. If we want to receive a specific piece of data, but we don't know if we can trust the sender, validating the data with a hash checksum or a digital signature is preferred. Security based on hash functions relies on the function producing a value from the data which would be infeasible to recreate with a different set of data. We can then compare the hash produced from the data with the validation hash. A digital signature allows us to verify that the data has been signed by a specific source using asymmetric key encryption.

Using checksums requires us to acquire the correct hash for the expected data beforehand by a trusted source. On the other hand, a digital signature can validate that an arbitrary data set is sent by a specific source but it requires us to have access to the sources public key.

Chapter 6
Solution Design

This chapter will provide a high level description of the solution, with the following chapter going into more detail and motivating the specific implementation choices.

Checksum Verification

The whole updating process and verification described so far poses multiple problems that need to be addressed. However, to address them all would be too large of a scope for this thesis. Therefore we will limit our scope somewhat. Specifically, we will make the assumption that the communication between the update-server and the clients is secure and everything that has been received from the server can be trusted.

A device will receive configurations from the trusted server. These configurations will contain references to services that the device should be able to fetch from any source that might have a copy of the service. This is where the first apparent security vulnerability in PalCom's updating scheme comes in. If clients should be able to accept services that come from any source, we need a method of ensuring that the services are authentic before execution. The only services that should be executed are those which are referenced by the current configuration, which has been received from the trusted server.

As we know from the previous section, using a secure hash function to produce a checksum for a file, and validating it against the expected sum, will make sure the file is the expected one. We receive the configuration from the server which then contains the references to the included services. We will therefore include the hash of the services in these references which means that we can trust that the validation hash is that of the correct file. We will calculate the same hash of the

service before activating it and compare this against the verification hash contained in the reference as depicted in fig. 6.1. This procedure will ensure that no corrupt services are ever allowed to execute. We will use Sha-256 for the checksum hashing since creating a malicious service that produces the same checksum as the correct one will require finding a second pre-image, which for Sha-256 currently is infeasible. An alternative solution would be to use a digital signature and let the server sign each service. But this alternative, would, however, require distributing keys and does not provide any extra security. Attacks such as Man-in-the-Middle do not need to be given any consideration in this case, as the service is already sent from an untrustworthy device. Additionally, as an attempt to detect corrupt services at an earlier stage, we will calculate the hash on a service and validate it with the hash contained in its own reference as soon as it is received.

In case a corrupt service is located, the client will remove it from its file system and later request to fetch a copy, preferably from another source. Future additions to this could include notifying other clients of a potentially corrupt service being circulated to avoid further spread.

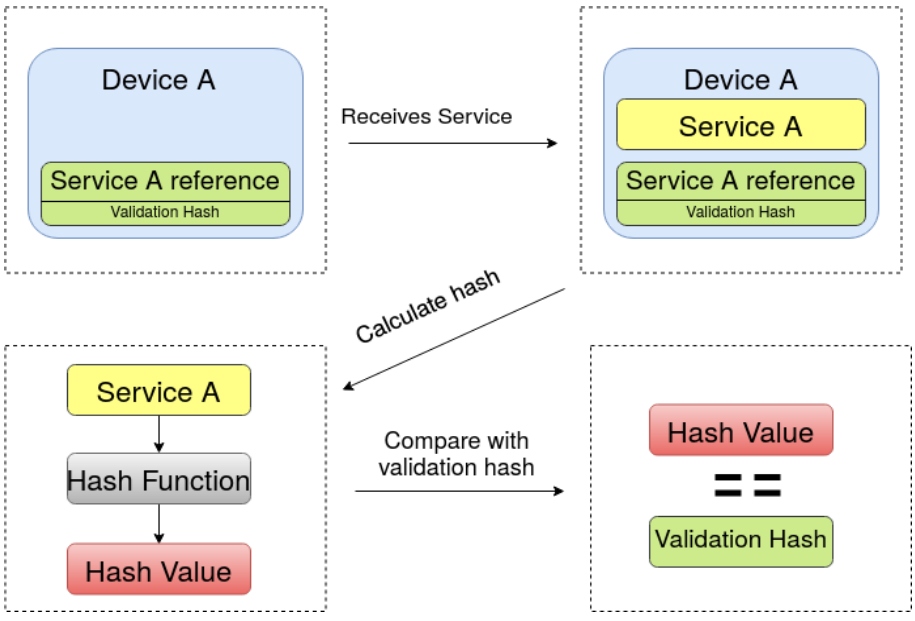

Figure 6.1: The verification process. Device A has a reference to a service which contains a validation hash. As A receives the service, the hash for the service is calculated using the hash function. The calculated hash is then compared with the validation hash from the service reference to verify that they match.

PAR-file Format

To provide a way to add additional information about services, such as compatibility issues, we introduce a new format: Par-files ('PalCom Archive' inspired by the java equivalent, Jar files). These will be the standard format used for representing a service or a configuration. Each Par-file is an archive file that consists of a main part and a corresponding metadata part. In the case of a service, the main part would be that service i.e. a jar file and the metadata will contain information such as compatibility and creator id. The metadata can be updated incrementally by users even after the main part itself is finished and released since they are separate from each other.

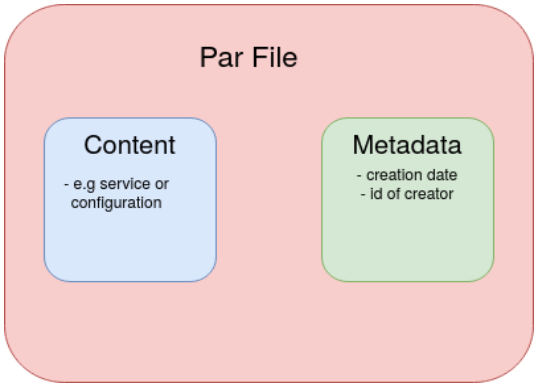

Figure 6.2: Par-file format, it includes the content file along with metadata information about the file

As this metadata could still contain crucial information, we will include a checksum verification for it. In order to allow additions to be made, the reference will specify a certain part of the metadata that it trusts and will only verify against this part. Any following information will be ignored but can be considered in a later distribution of the configuration when it has been verified to be trustworthy. Changing the configuration to include this new information could be seen as the trusted administrator approving or "signing" these additions with the new checksum. This allows users to freely add new information and still keep supplying the Par-files to other devices without causing their verification to fail.

Chapter 7
Implementation

This chapter will provide a more detailed description of the implementation details.

PalCom detailed description

To understand the implementation process, we will provide a more detailed description of the PalCom functionality, including names of classes and their function.

All incoming and outgoing communication regarding configurations and their included content by the configuration server is handled by a class called the `ConfigurationDBService`. It is implemented as a regular PalCom service and therefore any device that has PalCom installed can be used as the server. It is set up to communicate with the `ConfigurationManagerCommunicationService` that should be present on every device. Each device has a number of Managers that handle specific parts of the system, most relevant are the `ConfigurationManager` and `ResourceManager`.

- The `ResourceManager` is responsible for storing and retrieving resources from the devices, it is the only gateway to directly access files on the devices internal file system and its tasks include storing resources, retrieving resources, checking if a resource exists etc.
- The `ConfigurationManager` is responsible for the logic behind configurations. It handles receiving configurations and deploying these on the device and requests the resources contained in the configurations. It contains a reference to the `ResourceManager` so it can save the configurations to the file system. The `ConfigurationManager` also has a communication service, which handles communication to the server.

To inform a client that its binding has changed, the configuration server will send `DeviceConfigBindings` to it. The device will store these as pending and can use them to extract references pointing to all the resources it will need to acquire, i.e. the configurations and its services. It will then check with the `ResourceManager` if the resources contained in the references exist in its local file system already. This is done by retrieving the Kind and Type from each reference included in the bindings and using these to construct a unique file path. Each Kind will be a folder which in turn contains folders for each Type. These folders include an `_index` file which contains a Map of references paired with the corresponding file name of the correct version of the Type. The file system structure is visualized in figure 7.1. If a map

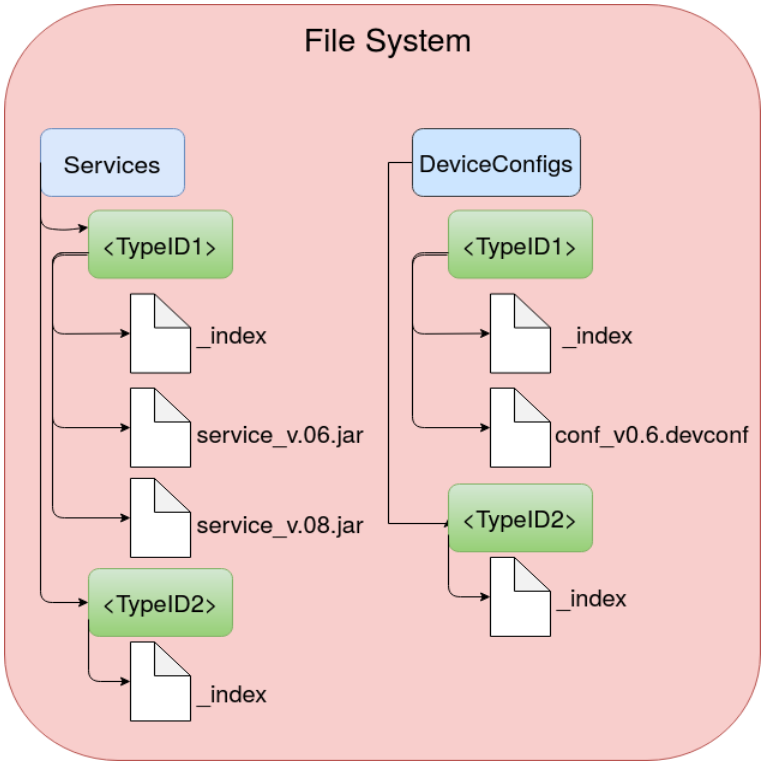

Figure 7.1: The file system structure. Each Kind will be represented by a folder. Each Type will be a subfolder. The index file will contain a mapping from references to the correct version of the file.

entry for a reference exists the system will conclude that the device has the resource already and move on. If the resource does not already exist a request will be sent to retrieve it and then the same process will be run again. When all resources have been acquired the device will load up the service and start to run on the new configuration. The communication hierarchy between the classes can be seen in figure 7.2. This means that all relevant code for retrieving and initiating services from the configurations is located in the `ConfigurationManager` class, and in some part of this process a verification step needs to be implemented to make sure malicious services get spotted.

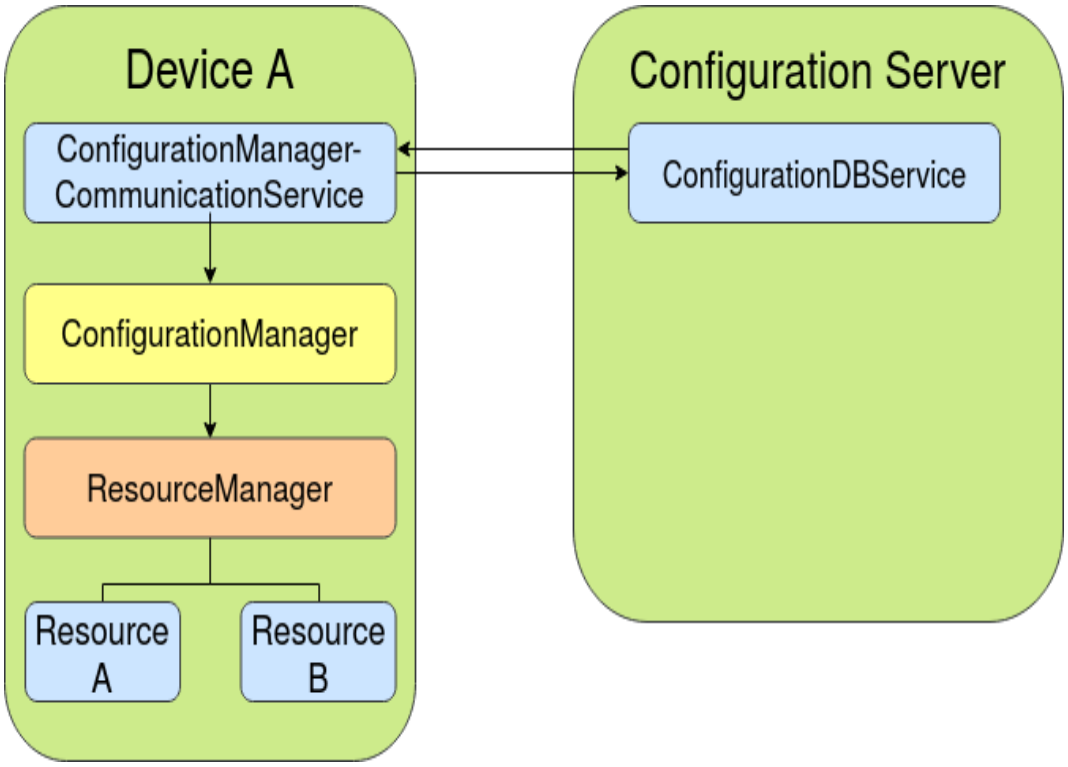

Figure 7.2: Communication structure, the server communicates with each client through the `ConfigurationManagerCommunicationService`. The internal class hierarchy communication is then as shown.

Checksum Validation

The configurations contain `ServiceTypeVersionReferences` pointing to the services, and the system constructs a unique file path based on the contents of the reference to find out where the referenced service should be located. The `ResourceManager` uses the function `getResourceFile` that takes a `TypeVersionReference` as parameter to retrieve the corresponding file. This function is used both for determining whether the resource exists and retrieving the resource for execution. The function will then retrieve the Kind and Type from the reference to find the correct folder for the index file (as described earlier). The key-value pair containing the references and service files is inserted when the resource is received from the source. This is done through the `addResource` function which takes a `TypeVersion` parameter. It will retrieve the reference to that `TypeVersion` by extracting it from the `TypeVersion` itself and use the Kind and Type in the same way to store the actual file contents in the file system.

The fact that the possibly malicious `TypeVersion` is able to provide its own reference for the mapping poses a security risk. An attacker could obtain a correct

`TypeVersion` for a service which is used in a configuration and then edit the jar file contents to whatever they like and distribute these to other clients. The reference extracted from this will then be unchanged and the client will believe that the file is the correct one. This specifically is why the verification checksum is crucial.

The malicious service will be detected when a configuration containing a reference to it arrives, i.e. before it is used. This should thus be safe since the code will in the case of a malicious service never be executed. We leave it to future work to handle available services that are not yet in use in a more optimal way.

The checksum validation step is implemented right before the device changes the configuration from pending to active. By this point it has acquired all the services and is about to start running them. Letting the `ResourceManager` handle the checksum calculation is reasonable as it handles all the system resources.

Listing 7.1: Checksum Calculation

```
public synchronized String getFileChecksum(File file) throws
   IOException {
       return getFileChecksum(file.getContents());
}
public synchronized String getFileChecksum(byte[] content)
   throws IOException {
       MessageDigest digest;
       try {
              digest = MessageDigest.getInstance("SHA-256");
       } catch (NoSuchAlgorithmException e) {
              return "";
       }
       ByteArrayInputStream bis = new ByteArrayInputStream(
          content);
       byte[] byteArray = new byte[1024];
       int bytesCount = 0;
       while ((bytesCount = bis.read(byteArray)) != -1) {
              digest.update(byteArray, 0, bytesCount);
       }
       bis.close();
       byte[] bytes = digest.digest();
       StringBuilder sb = new StringBuilder();
       for (int i = 0; i < bytes.length; i++) {
              sb.append(Integer.toString((bytes[i] & 0xff) +
                 0x100, 16).substring(1));
       }
       return sb.toString();
}
```

The `getFileChecksum`[7.1] function is placed in the `ResourceManager` and will, given a file or a byte array of the file contents, calculate its Sha-256 hash as a hex-

adecimal string. Note that the PalCom system uses its own File-class instead of the standard Java version and will include slightly different functions. We also added the function `validateResources` to the `ResourceManager` which simply takes a set of references and retrieves the referenced files and compares the produced hashes with the string validation hashes contained in the references. The function will return a set of all the references that had a mismatch in the hashes. This validation check will be performed in the `ConfigurationManager` just before deploying the configuration as seen below.

Listing 7.2: Resource validation

```
TypeVersionReference[] corruptRefs = resourceManager.
   validateResources(requiredResource)
                              .getTypeVersionReferences();
for (int i = 0; i < corruptRefs.length; i++) {
       TypeVersionReference current = corruptRefs[i];
       logInfo("Corrupt resource found in storage: The file
          referenced      ("+current.getUniqueName()+") does not
          match its
                              given checksum)");
       resourceManager.removeResource(current);
}
```

Testing

To test that the new functionality correctly performs its tasks and accepts services that match the validation hash and discards services that do not, we have set up some test scenarios.

In order to make sure that the validation functions themselves work correctly, a manual test is set up where services are placed in a simulated file-system manually beforehand. This works because when the `ResourceManager` only checks for existing services by using the information from the references it has to construct a file path to where the service should be located, and does not store any extra information if the services were added by the system itself. A mock file-system is created and given to a standalone instance of a `ResourceManager`. The system never communicates anything directly upwards from the `ResourceManager` so this is all the set up that is needed.

Three different services are placed in the file system, these are fully functional services that can run on the current version of PalCom. The `_index` files which contain the reference-service mapping are manually edited so that these references now contain the validation hash attribute corresponding to the service. This is needed because these references are compared to the actual ones when the system looks for existing services. The hash value for one service is left incorrect for testing purposes. Three configuration PON files are then edited to include hash values in

its references as well and are read directly from the test program to retrieve the corresponding `ReferenceSet` for testing the validation function. These three cases are now tested.

- A configuration containing a hash value that correctly matches the services
  This is the intended scenario for the system. The configuration references a service which currently exists in the file system and the service is in the correct state and should therefore be accepted by the system. This test passes without an exception.

- A configuration containing a hash value that does not match the services
  This represents a scenario where a malicious service has been distributed by an attacker. The service is located in the correct location as referenced by the configuration, but the file itself has been tampered with and produces a different hash from the expected one. In this case the system correctly throws an exception due to the produced hash and the validation hash not being equal.

- A configuration containing a hash value that does not match the services but the _index file contains a reference with a hash matching the services
  This represents a scenario where an attacker has edited the service contents and also edited the validation hash contained in the reference-service mapping to match the corrupted service. (The reference for the mapping is retrieved by the system from the service and is therefore also able to be tampered with). Here the system does not even recognize the file as existing since the reference from the configuration is no longer the same as the one in the map. Thus, no damage will be done.

Par-file implementation

In the current implementation, a `TypeVersion` includes a byte array with its content (e.g. a service or a configuration) and it is these classes in their PON format that are communicated between the devices. This means that no changes have to be made in regard to making the device communication work with the new Par format except for making the byte array contain the Par file instead. The handling of the archiving and extraction of the Par files needs to be kept consistent and we therefore introduce a new class `PARUtils` with static methods to create a par file out of a content part and its metadata part as well as methods for extracting the respective parts from the Par file. The Par archive itself does not require any unique functionality and to avoid reinventing the wheel we used the Apache Commons Compress[16] library to compress and extract the Par files using the same protocol as Tar files. These methods, compress and unPar (for compression and extraction), are listed below.

Listing 7.3: Compression

```
public static byte[] compress(byte[] file, byte[] meta,
   String filename) throws IOException {
```

```
            ByteArrayOutputStream bytes = new
               ByteArrayOutputStream();
            TarArchiveOutputStream out = new
               TarArchiveOutputStream(bytes);
            addToArchiveCompression(out, file, filename);
            addToArchiveCompression(out, meta, "metadata");
            out.close();
            return bytes.toByteArray();
      }

      private static void addToArchiveCompression(
         TarArchiveOutputStream out, byte[] content, String name)
                  throws IOException {
            TarArchiveEntry entry = new TarArchiveEntry(name);
            entry.setSize(content.length);
            out.putArchiveEntry(entry);
            out.write(content);
            out.closeArchiveEntry();
      }
```

Listing 7.4: Extraction

```
public static byte[][] unPar(byte[] file) throws IOException {

            TarArchiveInputStream myTarFile = new
               TarArchiveInputStream(new ByteArrayInputStream(file
               ));
            TarArchiveEntry entry = null;
            int offset;
            byte[][] files = new byte[2][];
            int counter = 0;
            while ((entry = myTarFile.getNextTarEntry()) != null)
               {
                  byte[] content = new byte[(int) entry.getSize()
                     ];
                  offset = 0;
                  myTarFile.read(content, offset, content.length
                     - offset);
                  files[counter] = content;
                  counter++;
            }
            myTarFile.close();
            return files;
      }
```

The parameters and return values are kept as byte arrays and the actual writing to the disk is handled by other classes. The `unPar` method returns an array of ar-

rays and its usage might be unintuitive and therefore extra methods to extract the content and metadata respectively as byte arrays have been added.

We will begin by implementing this format for the services. There needs to be a way of creating the Par files to begin with, which would be done when the service jar file has finished its development. The `ServiceManager` class is what handles the service management similarly to the resource and configuration managers and it seems reasonable to let it handle the initial creation of the service Par file. The `ServiceManager` has a service which allows users to interact with it remotely through the PalCom browser. This manager-service currently includes a command for adding a service to the file system, and its usage could be seen as a way of adding a finalized service to the file system of the configuration server. We will now add the Par archiving process to this command. It will, given a service jar file, package this into a Par file containing the service and an initialized metadata file. The format of the metadata will be that of a key value map, where the current protocol is to have a set of predetermined keys such as creation date and the device id of the publisher. To allow users to still add information to the metadata without breaking the verification, we will specify how many values to read from the metadata and provide a checksum that matches these values only. We will refer to this as lines even though each value is not necessarily stored as a different line in the format. Anything contained after these lines will be ignored, but can be considered in a later configuration containing a new checksum. The metadata checksum as well as how many lines to read is added to the references.

The current metadata format is stored as text, which is required to start with the version of the format followed by the hash value of the version number. This is to allow for backwards compatibility in case the format is changed later on and needs to be kept constant for future updates. An example of a metadata file containing only the version would be:

```
Version:1.0:b0453560c8c1ed6f44df6b5373fb2ddf
                    a950a07614c965588e9deaaf220c8c65
```

Following this will be a PON representation of a list including all the key value pairs. These are represented as `MetadataEntry` objects which include a key, a timestamp of when it was added, a byte array containing the value and a hash. This hash will be the hash of the line combined with the previous lines hash. You can think of it as the recursive calculation

$$hash_x = h(Line_x + hash_{x-1})$$

Where $h(x)$ is a hash function and $hash_0 = \epsilon$. This assures that any change in the text will also change all future hashes. The reason for this format is to allow clients to be able to retrieve the hash for any line without having to compute it every time. This can be used for verifying that the data is correct before sending it to another client that has requested it. As the values are stored as byte arrays, they

can originally be anything from a string to a PON object and it is up to the parser to interpret this correctly depending on the key, and convert the values accordingly. To work with the metadata we have added a `Metadata` class to represent it, which can be retrieved from a Par file by calling a function in the `PARUtils` class. This class includes functionality for parsing the text format into a list attribute for all key value pairs. We represent this map structure as a list since it is important that each line must stay in the order of their creation time so that the hash values never change. Individual get-methods and set-methods will need to be added for each new attribute as they are introduced and these will then iterate through the list and look for the correct key, and then convert the value to the correct class. All values will be returned as a `MetadataValue` class that includes both the value and the timestamp. To ensure that the metadata does not return values that have not been verified with the checksum, it will store a maximum line number for how far in the list it will search. This can be set with a function call and should always be the same as what is included in the reference.

Since the content from which to calculate the checksum on is now contained in this `Metadata` class, we introduce a new interface `ChecksumCalculator` that only consist of a method to calculate the checksum of a byte array. We change the `ResourceManager` to include an instance of a `Sha256Calculator` to use for its calculations. We then let the `Metadata` calculate its own checksum by creating a method that takes a `ChecksumCalculator` and a line number and then returns the resulting hash string from doing the iteration, hashes from the entries are calculated by adding the key, value and timestamp together with the previous line hash. Add-methods will also require a `ChecksumCalculator` parameter as it also adds the hash automatically for each new key value pair. The metadata is also separate from the actual Par file stored on the disk, meaning that adding values to the metadata class does not directly influence the file itself. To save any changes to disk we have added a `updateMetadata` function to the `ResourceManager` that given a reference to a Par file, will update its metadata file to the contents of a given metadata class. When an instance of this `Metadata` class is created, it will make sure the format is correct and all the internal hashes are correctly generated, otherwise it will throw an `InvalidFormatException`. This enables us to detect potentially corrupt files before having a reference from a configuration for comparison and will prevent further spread. The `ResourceManager` will now validate the content file and metadata file separately using the checksums and line count contained in the references.

Chapter 8
Evaluation

To evaluate how the implementation functions in practice, we will return to the nurses' tablet scenario and evaluate the results. This will be done by using the included PalCom tools to simulate a real world interaction of the system. We start by creating a new device to act as a client (a nurses private tablet in our scenario). We then create another device that will act as the configuration server. We use an example service to act as the newly patched patient record software and package this into a Par-file for distribution. Since the PalCom system currently does not include functionality for receiving services from other clients we will let the tablets retrieve these from the server. We then create a new configuration that includes this service and bind it to the tablet device. The server now informs the tablet of the change and makes it retrieve the new service from the server. This is performed successfully and the tablet now has the updated service software in its file system and can start running the new configuration. Since the service was accepted, this means that the checksums for the content and metadata correctly matched up with the references checksums and that the verification is working as intended. We will make sure that it also detects erroneous situations by testing a scenario in which an attacker intercepts the communication and provides a corrupt service. Implementing an actual attack would be unnecessarily advanced, and since the client to client requesting is not yet implemented in the system, we will simply manually edit the par service file stored in the configuration server. This results in that the client correctly concludes that the received service does not match its checksum and does not attempt to execute it.

The overall process is working as intended but there are a few inconveniences. Since the GUIs are currently under development, using them together with each other can be inconvenient. Mainly, since the Configuration server is set up as a separate service that is running on a device, we cannot package a service into a par-file directly onto the configuration server and we need to package it separately before importing it to the server. When detecting a corrupt service there remains to design the

handling of the situation to avoid requesting the same corrupt service again. One option would be to always retrieve a resource from the trusted update-server after it has been detected to be corrupt. Additionally, one could mark the source of the corrupt service as untrustworthy and avoid it in the future.

The authentication process will remain secure as long as finding a second preimage for sha-256 remains infeasible. As sha-256 is a fast hash function the extra computational time will be negligible even for smaller devices. The Par-file format functions as intended and can be stored and retrieved from disk while still generating the same checksum. However, changes to the metadata functionality will likely have to be made after discovering new requirements from real life use. Future additions would include extending the par format for other kinds, such as configurations.

Our solution does not use any stored keys for authenticating the updates which differs from how for example the Microsoft Windows update client works. The act of signing updates is replaced with including the updates checksum in the configuration. This means that it is crucial that the communication with the configuration server is secure since we have no way of detecting a corrupt update apart from that checksum. As Windows automatic software updating system is posed with a similar problem of authenticating the updates, their solution is instead based on Microsoft signing the code which is then verified by the client computer. This does add an extra level of certainty since even if all outside communication managed to be infiltrated, corrupt software could never pass the verification.

Chapter 9
Conclusion

Any system using this PalCom implementation will be secure against having malicious software pose as a new update. This is achieved by an authenticated server providing the software's checksum that will be used for validation before the software is executed. This allows for safe deployment of new updates while still allowing the system's devices to distribute updates internally without the risk of corrupt software being allowed to execute. The Par-file format allows for additional information such as testing and compatibility to be added to services after publishing them while still passing the verification process. Each Par-file will consist of a content file and its metadata separated from each other and these will be verified separately, making it possible to incrementally update the metadata.

References

[1] L. Courtès, "Functional Package Management with Guix", Bordeaux, France *https://arxiv.org/pdf/1305.4584v1.pdf*

[2] Felser M., Kapitza R., Kleinöder J., Schröder-Preikschat W. (2007) Dynamic Software Update of Resource-Constrained Distributed Embedded Systems. In: Rettberg A., Zanella M.C., Dömer R., Gerstlauer A., Rammig F.J. (eds) Embedded System Design: Topics, Techniques and Trends. IFIP – The International Federation for Information Processing, vol 231. Springer, Boston, MA *https://link.springer.com/content/pdf/10.1007/978-0-387-72258-0_33.pdf*

[3] Kevin Dunn, Automatic update risks: can patching let a hacker in?, 29 July 2004 *https://www.sciencedirect.com/science/article/pii/S1353485804001023*

[4] L. Kvarda, P. Hnyk, L. Vojtech, Z. Lokaj, M. Neruda, T. Zitta, "Software Implementation of a Secure FirmwareUpdate Solution in an IOT Context", Czech Technical University in Prague, *https://pdfs.semanticscholar.org/7515/43344eee3b7716e4fbcdcf57c6520de0afe0.pdf*

[5] B. Magnusson, M. Nordahl, "A lightweight data interchange format for internet of things with applications in the PalCom middleware framework" 14 May 2016 *https://doi.org/10.1007/s12652-016-0382-3*

[6] itACiH *https://itacih.se/*

[7] M. Weißbach, N. Taing, M. Wutzler, T. Springer, A. Schill and S. Clarke, 2016 IEEE 3rd World Forum on Internet of Things Service A(WF-IoT), Reston, VA, 2016, pp. 171-176. doi: 10.1109/WF-IoT.2016.7845450 *https://ieeexplore-ieee-org.ludwig.lub.lu.se/document/7845450*

[8] M. Yang and F. Zhu, "The Design of Remote Update System Based on GPRS Technology," 2010 International Conference on Management and Ser-

vice Science, Wuhan, 2010, pp. 1-4. doi: 10.1109/ICMSS.2010.5575699 *https://ieeexplore.ieee.org/document/5575699*

[9] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. Bianco, C. Baisse "Announcing the first SHA1 collision", 23 February 2017, Google Security Blog *https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html*

[10] H. Gilbert and H. Handschuh, "Security Analysis of SHA-256 and Sisters" , 2004 *https://link.springer.com/content/pdf/10.1007%2F978-3-540-24654-1_13.pdf*

[11] J. Lane Thames, R. Abler and D. Keeling, "A distributed active response architecture for preventing SSH dictionary attacks," IEEE SoutheastCon 2008, Huntsville, AL, 2008, pp. 84-89. doi: 10.1109/SECON.2008.4494264 *https://ieeexplore.ieee.org/abstract/document/4494264*

[12] M. Månsson, "Dynamic installation and automatic update of Bluetooth low energy devices in PalCom", 8 June 2015, Department of Computer Science, Lund University.

[13] B. Magnusson, B. Johnsson, G. Hedin, "Factoring out Glue-code in Systems of IoT Devices", Department of Computer Science, Lund University.

[14] M. Nordahl A. Åkesson, B. Magnusson, G. Hedin, "Software updates and maintenance for IoT", Department of Computer Science, Lund University.

[15] B. Magnusson, G. Hedin, P. Runeson, "PalCom MIST: a Metaprotocol for Internet Systems of Things", Department of Computer Science, Lund University. A. Bellissimo, J. Burgess, K. Fu, "Secure Software Updates: Dissapointments and New Challenges", Department of Computer Science, University of Massachusetts Amherst. *https://spqr.eecs.umich.edu/papers/secureupdates-hotsec06.pdf*

[16] Apache commons compress library *https://commons.apache.org/proper/commons-compress/*

Examensarbete Secure Updating of Configurations in a System of Devices
Handledare Mattias Nordahl (LTH)
Examinator Boris Magnusson (LTH)
Populärvetenskaplig Sammanfattning

Säker uppdatering av konfigurationer i ett digitalt system

Jens Mellberg
Instutitionen för datavetenskap
Lunds tekniska högskola

10 februari 2020

Digitala system bestående av en stor mängd enheter blir alltmer vanligt förekommande i takt med att vårt samhälle fortsätter att digitaliseras. Dessa enheter behöver uppdateras regelbundet och för enkelhetens skull görs detta ofta över internet. Detta öppnar upp en möjlighet för "hackers" att tillverka falska uppdateringar som de kan lura enheterna att installera. Detta arbete går främst ut på att ta fram en metod för att förhindra att dessa falska uppdateringar godkänns av systemet.

Detta arbete har utförts på PalCom-ramverket som är utvecklat på Lunds Universitet och det används som bas på IoT-system för att hantera kommunikationen mellan enheterna. Ett exempel på ett sådant system är det smarta hemmet, som kan involvera allt från hemlarm till värmesystem. Programvaran i dessa system behöver ibland uppdateras, ofta på initiativ av leverantören. PalCom sköter godkännandet och installeringen av den uppdaterade mjukvaran. Detta arbete konstruerar en lösning för PalCom som kan upptäcka skadlig programvara vid uppdateringar och avfärda dessa. PalCom förser systemets enheter med konfigurationer, dessa beskriver vilka programvaror som enheten ska köra. Dessa programvaror kan hämtas från andra enheter i systemet och måste då verifieras som autentiska. Den föreslagna lösningen bygger på att använda kontrollsummor för att verifiera autenticiteten. Dessa kontrollsummor kommer att inkluderas i konfigurationerna. En algoritm används för att producera en sådan kontrollsumma baserat på hur programvaran ser ut. Om någon skulle ändra programvaran kommer algoritmen producera en annan kontrollsumma. Enheterna kan då jämföra kontrollsumman i konfigurationen med den producerade kontrollsummen varje gång en uppdatering utförts och på så sätt upptäcka om felaktig programvara mottagits. Detta är en teknik som används ofta och säkerheten i metoden bygger på att dessa kontrollsummor ska vara svåra att förfalska. Mer specifikt så ska det vara svårt att skapa falsk programvara som producerar samma kontrollsumma som den äkta programvaran.

Under arbetet har det även utvecklats en metod för att kunna lägga till information om en uppdatering efter att den har börjat användas. När en uppdatering har installerats och börjat användas i ett system kan man som administratör vilja "tagga" programvaran med information som t.ex. kompabilitet med andra delar i systemet eller stämpla den som godkänd när den har testats. Vi vill kunna göra detta utan att ändra programvaran i sig, eftersom det skulle leda till att en annan kontrollsumma produceras och programvaran avvisas.

Arbetet har resulterat i en lösning som kan garantera att den programvara som mottas är korrekt i system som använder PalCom. Lösningen kommer högst sannolikt även att vara säker för en lång tid framöver baserat på säkerheten hos algoritmen som producerar dessa kontrollsummor. Även efter att den användna algoritmen har blivit utdaterad kan den enkelt ersättas med en mer säker algoritm. Detta arbete kan potentiellt bli viktigt i säkerhetskritiska applikationer som hemlarm och inom industri. Arbetet beskrivs närmare i rapporten: Jens Mellberg, "Secure Updating of Configurations in a System of Devices", Department of Computer Science, Lund University, Sweden, 2020, ISSN 1650-2884 LU-CS-EX 2020-04.