Using Hash Trees for Database Schema Inconsistency Detection

CHARLOTTA SPIK
Using Hash Trees for Database Schema Inconsistency Detection

CHARLOTTA SPIK
spik@kth.se
2019-06-27

Master thesis, 30hp

Examiner: Vladimir Vlassov
Academic Supervisor: Johan Montelius
Industrial Supervisor: Denys Knertser

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science (EECS)
Department of Software and Computer Systems
SE 164 40 Kista, Sweden
Abstract

This work was carried out for Cisco Systems Inc., a global IT and networking company that develops products and services related to networks and telecom. Devices developed by Cisco are configured using a device configuration service, and the configurations are stored in a configuration database. When a device configuration is updated, the database schema for that device is updated. When this happens, the new and old version of the schema need to be compared for inconsistencies in order to find which elements of the schema need to be updated. This process is very slow as it is done today, and the time this takes affects the customers directly.

The purpose of this project is investigate if the performance of the inconsistency detection process can be improved by developing an algorithm based on Merkle trees. Based on previous work related to the area, the hypothesis is that this will considerably improve the performance.

Merkle trees are hash trees that allow hashes of data to be compared to each other from the root down to the leaves. Only branches where some difference in data has occurred need to be traversed. This avoids having to iterate through the whole schema. Each schema version will be represented by a hash tree. When a new version of a schema enters the system, a hash tree will be built for this version. The hash trees of the previous and the new version of the schemas will be compared to find possible inconsistencies.

For this work, two algorithms have been developed to improve the performance of the inconsistency detection by using Merkle trees. The first builds a hash tree from a database schema version, and the second compares two hash trees to find where changes have occurred. The results of performance testing done on the hash tree approach compared to the current approach used by Cisco where all data in the schema is traversed, shows that the hash tree algorithm for inconsistency detection performs significantly better than the complete traversal algorithm in all cases tested, with the exception of when all nodes have changed in the tree. The factor of improvement is directly related to the number of nodes that have to be traversed for the hash tree, which in turn depends on the number of changes done between versions and the positioning in the schema of the nodes that have changed. The real-life example scenarios used for performance testing show that on average, the hash tree algorithm only needs to traverse 1.5% of the number of nodes that the complete traversal algorithm used by Cisco does, and on average gives a 200 times improvement in performance. Even in the worst real-life case used for testing, the hash tree algorithm performed five times better than the complete traversal algorithm.

Keywords

Merkle Tree, Hash Tree, inconsistency detection, Anti-Entropy Repair, replica synchronization
Referat


Syftet med detta projekt är att undersöka om det går att förbättra prestandan för att upptäcka skillnaderna mellan scheman genom att utveckla en algoritm baserad på Merkle träd. Baserat på tidigare arbeten relaterad till området är hypotesen att detta skulle lösa problemet genom att öka prestandan betydligt.


I detta arbete har två algoritmer utvecklats för att förbättra prestandan på processen att hitta skillnader mellan scheman genom att använda Merkle träd. Den första bygger ett hashträd från schemaversionen och den andra jämför två hashträd för att hitta var förändringar har skett. Resultaten från prestandautvärderingen som gjorts på hashträdalgoritmen jämfört med nuvarande algoritm som används på Cisco där all data i schemat traverseras, visar att hashträdalgoritmen presterar signifikant bättre än algoritmen som traverserar all data i alla fall som testats, förutom då alla noder har ändrats i trädet. Förbättringsfaktorn är direkt kopplad till antalet noder som behöver traverseras för hashträdalgoritmen, vilket i sin tur beror på antalet förändringar som skett mellan versionerna och positioneringen i schemat av de noder som har förändrats. De exempelscenario som har tagits från riktiga uppdateringar som har skett för existerande scheman visar att i genomsnitt behöver hashträdalgoritmen bara traversera 1,5% av noderna som den nuvarande algoritmen som används av Cisco måste traversera, och hashträdalgoritmen ger i genomsnitt en 200 gånger prestandaförbättring. Även i det värsta fallet för dessa uppdateringar tagna från verkliga scenarier presterade hashträdalgoritmen fem gånger bättre än algoritmen som traverserar all data i schemat.

Nyckelord

Merkle träd, Hashträd, detektion av inkonsekvenser, Anti-Entropi Reparation, Synkronisering av replikor
ACKNOWLEDGEMENTS

I would first like to thank my thesis supervisor at KTH, Johan Montelius. Johan has provided a lot of help and support throughout the project. He was always available for a chat and has spent much time meeting with me to discuss issues and provide feedback on the work and report. For this, I am incredibly grateful.

I would also like to thank professor Vladimir Vlassov from KTH, who served as the examiner for this work. He has provided very valuable feedback on the report that has led to many improvements, and this is much appreciated.

I would also like to acknowledge Denys Knertser, who was my supervisor at Cisco, and who I want to thank for the time he put in on meetings to explain internal systems used at Cisco and to discuss any questions I had. In addition, I want to extend my gratitude to the everyone in the Tail-f department at Cisco where I worked during the thesis, for being so welcoming. Everyone was always ready and willing to answer any questions and offer support when needed. Thank you all for making me feel like part of the team.

And finally, I would like to thank Isabel Ghourchian, my classmate, colleague and very close friend, for spending much of her very limited time offering feedback and insight on my work every step of the way, and for being an incredible support throughout not only this thesis work, but through my years at KTH.
# Table of Contents

**Introduction** ........................................................................................................ 1
  1.1. Background ........................................................................................................ 2
  1.2. Problem ............................................................................................................... 2
  1.3. Purpose ............................................................................................................... 2
  1.4. Goal .................................................................................................................... 3
  1.5. Benefits, Ethics and Sustainability .................................................................... 3
  1.6. Research Methodology ....................................................................................... 3
  1.7. Contributions ..................................................................................................... 5
  1.8. Delimitations ...................................................................................................... 5
  1.9. Outline ............................................................................................................... 6

**Background** .......................................................................................................... 7
  2.1. Relevant Systems ............................................................................................... 7
  2.2. Merkle Trees and Hash Trees ............................................................................ 10
  2.3. Related Work .................................................................................................... 13
  2.4. Summary ........................................................................................................... 15

**Methodology** ......................................................................................................... 16
  3.1. Research Process ............................................................................................... 16
  3.2. Research Paradigm ............................................................................................ 17
  3.3. Planned Measurements ...................................................................................... 19

**System Architecture** ............................................................................................. 21
  4.1. Hashing .............................................................................................................. 21
  4.2. Hash Tree Structure .......................................................................................... 24
  4.3. Inconsistency Detection ..................................................................................... 26
  4.4. Performance Testing .......................................................................................... 29

**Results and Analysis** ............................................................................................ 31
  5.1. Specially Constructed Cases .............................................................................. 31
  5.2. Real-Life Update Scenarios .............................................................................. 35
  5.3. Discussion ......................................................................................................... 44

**Conclusions and Future Work** ............................................................................. 46
  6.1. Conclusions ....................................................................................................... 46
  6.2. Limitations ........................................................................................................ 47
  6.3. Future Work ...................................................................................................... 48

**References** ........................................................................................................... 49
List of Figures

Figure 1: YANG tree structure for example module "System"...............................10
Figure 2: Merkle tree structure example .................................................................11
Figure 3: Example of hash tree structure ..............................................................25
Figure 4: Tree of changes returned from inconsistency detection algorithm ...........29
Figure 5: Performance of hash tree algorithm compared to complete traversal algorithm when no changes has occurred between versions .............................................32
Figure 6: Performance of hash tree algorithm compared to complete traversal algorithm when one change to a leaf near the root has occurred between versions ........33
Figure 7: Performance of hash tree algorithm compared to complete traversal algorithm when one change to a leaf node in a deep branch has occurred between versions ..........34
Figure 8: Performance of hash tree algorithm compared to complete traversal algorithm when all nodes have changed between versions .................................................35
Figure 9: Real-life update scenarios for schema A............................................39
Figure 10: Real-life update scenarios for schema B..........................................40
Figure 11: Real-life update scenarios for schema C..........................................40
Figure 12: Real-life update scenarios for schema D...........................................41
Figure 13: Real-life update scenarios for schema E...........................................41
Figure 14: Real-life update scenarios for schema F...........................................42
Figure 15: Relationship between time and number of nodes traversed by the hash tree algorithm..............................................................................................................42
Figure 16: Relationship between time and number of nodes traversed by the hash tree algorithm, last value excluded ..........................................................43
List of Tables

Table 1: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema A.................................................................36
Table 2: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema B.................................................................37
Table 3: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema C.................................................................37
Table 4: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema D.................................................................38
Table 5: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema E.................................................................38
Table 6: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema F.................................................................39
Table 7: Average number of nodes traversed for complete traversal and hash tree approach as well as the factor of improvement in number of nodes to traverse for each schema........43
Table 8: Average traversal time for complete traversal and hash tree approach as well as the factor of improvement in time for each schema.................................................................44
## Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NSO</strong></td>
<td>“Network Services Orchestrator”. Cisco’s device configuration service.</td>
</tr>
<tr>
<td><strong>CDB</strong></td>
<td>Cisco’s configuration database where device configurations are stored.</td>
</tr>
<tr>
<td><strong>YANG</strong></td>
<td>Data modeling language used by Cisco to describe database schemas.</td>
</tr>
<tr>
<td><strong>SHA</strong></td>
<td>“Secure Hash Algorithms”. A family of cryptographic hash functions.</td>
</tr>
<tr>
<td><strong>MD4/MD5</strong></td>
<td>“Message Digest” 4 and 5. Cryptographic hash functions.</td>
</tr>
<tr>
<td><strong>RIPEMD</strong></td>
<td>“RIPE Message Digest”. A family of cryptographic hash functions.</td>
</tr>
<tr>
<td><strong>Erlang</strong></td>
<td>Functional programming language used for development of prototype.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This project was carried out for Cisco Systems Inc. Cisco is a global company developing products and services related to IT, networks and telecom. The Tail-f department of Cisco develops software automation services for networking hardware. One software developed by Tail-f is called NSO, and is used for automating device configurations. These configurations are stored in a database called CDB according to schemas defined in YANG. When a device configuration changes at the customer, for example when a driver is updated, the YANG schemas are updated. When this happens, the old version of the schema needs to be compared to the new. The schema cannot just be replaced as the changes need to be approved. Therefore, both versions of the schema need to be iterated through in order to find where the inconsistencies between them lay, so that the updates can be made. This procedure is costly due to the fact that the schemas can be very large, and iterating through them therefore takes considerable time.

This work aims to investigate whether an algorithm inspired by Merkle trees can improve the performance of the schema inconsistency detection. Merkle trees are hash trees where each leaf node is a hash of a data block, and each non-leaf node is a hash of its children. Therefore, when comparing two different Merkle trees with each other, it can be determined that if two node hashes differ between the two trees, there is a difference in the hashes of the node’s children. The tree can then be traversed from the root down, constantly checking what node hashes have changed, until one or more data blocks that the tree is built on are reached. These data blocks are then the data blocks that differ between the two trees.

Merkle trees are used in for example Dynamo, Apache Cassandra and Riak. All these are distributed key-value stores that use replication of data for failure handling. These distributed stores use Merkle trees for detecting inconsistencies of data between different replicas efficiently. It is therefore possible that a hash tree algorithm inspired by Merkle trees and adjusted for the purpose of detecting inconsistencies between different versions of YANG schemas can be used to improve the performance of schema inconsistency detection. The hypothesis is that by using hash trees, the comparison will be more efficient since only branches of the tree where changes have occurred need to be traversed and compared. Only if every element has been changed must every branch be iterated through. Since the need to
compare each component in the schema is eliminated, this has the potential of improving the performance considerably.

This work will implement a prototype of a hash tree algorithm for schema inconsistency detection and perform performance tests to compare the performance of this algorithm with an algorithm built to simulate the current algorithm used by Cisco today, which traverses all elements always, in order to determine if a hash tree algorithm is indeed more effective.

1.1. Background

Cisco Systems is the worldwide leader in networking for the internet[1]. They manufacture both products and services related to networking, telecom and IT[2]. Cisco uses NSO, Network Services Orchestrator, to handle device configurations. NSO stores the device configurations in a configuration database called CDB with predefined schemas defined by YANG. NSO was created by Tail-f, which was acquired by Cisco in 2014[3]. Tail-f develops software for networks and network devices, and is the leader in network programmability and data model-driven device management[4]. NSO is now sold as a service by Cisco Tail-f to other companies in need of a system to automate network service configurations. More information about NSO, CDB and YANG can be found in section 2.1.

When a customer adds, removes or changes the configuration of a device, for example a driver, in NSO, this sparks an update of the YANG schema describing that device. The schema must then be updated to the newer version in CDB. When performing the update, the new and old version of the schema must be compared in order to find changes between the two versions. If the changes that were found are allowed updates, the schema in CDB is updated so that CDB contains the latest version of the schema.

1.2. Problem

The current approach to perform the YANG schema inconsistency detection used by Cisco is to iterate over both versions of the YANG schema descriptions to determine where the inconsistencies lay between the schema versions according to some predefined rules. This is an expensive procedure as the schema descriptions can be very large, and the schema descriptions are compared linearly to each other. This gives the process a time complexity of O(n), where “n” is the number of elements in the YANG schema. This makes the operation highly ineffective.

The schema descriptions can and often do contain hundreds of thousands of elements that need to be compared, where each element contains several fields containing some data about the element. It can take several hours from when the customer starts the update procedure to the time the update is completed in the system. This operation affects the customers directly, so performing a device update means that a customer will have to wait until the process is finished.

1.3. Purpose

The purpose of the thesis is to investigate whether a hash tree algorithm inspired by how Merkle trees are used in for example Dynamo and Apache Cassandra for inconsistency detection can be used to improve the performance of inconsistency detection between different versions of a database schema description. If it is determined that this can be done, Cisco can use the algorithm in their current system in order to improve its performance,
thereby speeding up the time it takes for the customer to update the schema. If it is determined that the algorithm will not improve the performance, Cisco can choose to either continue using the current system assuming that it cannot be sped up, or can investigate alternative algorithms to speed up the process. Furthermore, the investigation performed for this thesis will add to the overall information that exist about hash trees and Merkle trees for inconsistency detection, thereby contributing to the overall information base that exist about this subject.

1.4. Goal

The goal of the project is to evaluate whether hash trees can be used for improving the performance of inconsistency detection between database schema versions. This also includes an evaluation of how much, if at all, the performance can be improved. This in turn includes determining the performance difference in best, worst and typical case scenarios. In this work, best case scenario would be if no changes have occurred to the schema, as for hash trees, if no change is detected in the hash of the root, the trees are the same and no traversal is required. The worst case scenario would be if all nodes have changed, as this means that all nodes in the tree need to be traversed to find all the differences. The typical case shows how the algorithm can be expected to perform in typical real-life scenarios. This will be determined by using real-life update scenarios. When these different scenarios have been investigated, it can be determined whether a hash tree algorithm can be expected to improve the performance of schema inconsistency detection when used in the current overall system at Cisco.

In order to reach the goal of the project, a prototype inconsistency detection algorithm must be developed so that it can be compared to the total traversal approach used by Cisco currently. This will include an algorithm that builds a hash tree from the database schemas, and an algorithm that performs the inconsistency detection. Finally, the prototype hash tree inconsistency detection must be performance tested compared to the total traversal approach to determine the change in performance.

The expected result of this is to have an answer to whether a hash tree algorithm based on Merkle trees will improve the performance of the system.

1.5. Benefits, Ethics and Sustainability

A system with low performance uses up more data resources and time, and thereby consumes more energy than a high performing system. Energy consumption is an important aspect for environmental sustainability. As it is now, the algorithm used by Cisco means that an operation made by customers can take hours of computational time. If this were to be significantly reduced, this would save a substantial amount of resources, which saves energy.

1.6. Research Methodology

In order to answer the question of whether hash trees can be used to improve database schema inconsistency detection, a prototype algorithm will be developed and several benchmarks will be performed. This involves designing and developing a prototype of the Merkle tree inconsistency detection algorithm and comparing the performance of this prototype to the current inconsistency detection approach used by Cisco, that performs complete traversal of the schemas. To determine the difference in performance, several performance tests using real-life examples of YANG schema updates will be performed.
The prototype will include an algorithm for building a hash tree from a YANG schema using a hashing algorithm developed for this purpose, and an inconsistency detection algorithm that takes two versions of a schema as input and determines what elements of the schemas differ between them. Because of time constraints, the developed system will not be integrated with the current Cisco system. Therefore, an algorithm mimicking the current Cisco approach for inconsistency detection will be developed so that performance testing can be done on this complete traversal algorithm compared to the hash tree algorithm. This way, the factor of performance difference between the two systems can be determined, which will show how much time, if any, the new hash tree algorithm could save compared to the complete traversal algorithm used by Cisco.

From this, it can be determined that the following must be done to achieve the goal stated in section 1.4 above:

- Determining how the hashing should be done; what hash algorithm to use, what data to hash and how to perform the hashing of the tree.
- Implementing a prototype algorithm that builds the hash tree from a YANG schema description.
- Implementing a prototype algorithm that performs the inconsistency detection between versions of the schemas using hash trees.
- Implementing an algorithm simulating the complete traversal approach to schema inconsistency detection used by Cisco in order for a comparison to be made between this approach and the hash tree approach.
- Implementing a performance testing algorithm that can be used to perform benchmarks on the hash tree algorithm and complete traversal algorithm with different schemas.
- Performing experiments for both versions of inconsistency detection on specially constructed best and worst case scenarios as well as real-world update scenarios faced by Cisco customers to determine the performance difference between the hash tree algorithm and the complete traversal algorithm. This will give a best, worst and typical case estimate of the performance difference.

The project work is inspired by the engineering design process. This is the process used by engineers when designing products or services, and consists of finding a need, coming up with possible solutions, performing research, designing the product, developing the product and testing the product before it is released. The steps to follow differ in literature depending on authors, but the general process is as described above. The process can be iterative, so some steps can be repeated in several iterations in order to make alterations and improvements to the product.

The engineering design process is often used when developing a product where the result is the release of the finished version of that product. In this case, the expected result is not a finished product but rather an answer to a question. However, a prototype will be developed in order to answer this question, and for developing this prototype the engineering design process can be used as a basis with some alterations made to the process. For developing this prototype, research and planning had to be performed to gather sufficient information and to simplify the development process respectively, before the actual implementation began. Testing is of course an important part as the prototype need to work as intended in order to give a useful result.

The engineering design process is described in more detail in section 3.2, and how it has been applied in this project work is shown in section 3.1.
1.7. Contributions

For this work, an algorithm that converts existing tree data structures into hash trees while preserving the structure of the original tree, as well as an algorithm that compares two hash trees to determine how they differ were developed.

The algorithm building the hash tree traverses the existing tree structure down to the leaves, and then converts all nodes in the tree to hashed representations of the nodes from the leaves to the root. The algorithm uses a hash for each node for detecting if the node itself has changed in any way by hashing the node data. The algorithm also uses another hash for each node to detect if there are any changes to the node’s subtree. This hash is a combination of the hashes of a node’s children. The children hashes are combined by using bitwise XOR. This is in contrast to how Merkle trees are usually built when joining the hashes, the reason behind this being that in the case of this work, a reordering of children nodes should not constitute as an inconsistency. As XOR is commutative, two sibling nodes changing position will not affect the end result of combining the child hashes.

The inconsistency detection algorithm compares two hash trees from the root down to the leaves and reports which nodes have changed, been deleted or been added to the tree. The algorithm matches the names of nodes from one tree to the nodes in the other tree in order to compare corresponding nodes to each other even if nodes have been reordered between siblings. This too means that sibling reordering will not be detected as an inconsistency.

The hash tree inconsistency detection algorithm has been performance tested compared to an algorithm that goes through all nodes in the original tree structure to find inconsistencies. The performance improvement has been determined to depend on the number of changes made to the tree and the placement of the changes in the tree for the hash tree algorithm, and the number of nodes in the tree for the complete traversal algorithm. For the trees used in this work, the algorithm shows an average 200 times improvement in performance when using the hash tree algorithm, and that in typical cases only 1.5% of the total number of nodes need to be traversed for the hash tree algorithm compared to the complete traversal algorithm.

For this work, a study on hash trees and Merkle trees has been done to gather information on the area, and the findings are presented in this report. From this, the idea of Merkle trees has been adapted for this work to create an alternative form of hash tree. The performance tests have shown how hash trees can improve the performance of inconsistency detection on data that is subject to change. This work has therefore also contributed to the overall information base that exists on Merkle and hash trees, how they can be used and how they can be adapted to different scenarios.

1.8. Delimitations

In this project, only one possible solution, using hash trees for inconsistency detection, will be investigated. There are likely more possible ways of improving the performance of the system, and investigating and implementing these would allow for a comparison between numerous solutions so that the best one could be picked, and the problem formulation could then be to investigating possible solutions to inconsistency detection. However, this would require more time than is available for the thesis, and is therefore out of scope for this thesis.

The applications built for this project will be implemented as independent applications from the existing system at Cisco, but will later be included in this system as system components.
CHAPTER 1: INTRODUCTION

The reason for this choice is twofold. Firstly because the algorithm used at Cisco for inconsistency detection performs the updates as soon as an update is detected. This means that it would be challenging to measure only the inconsistency detection time as this would have to be separated from the update time. Secondly because it would require significant time to investigate the existing system in order to write code that can be integrated and used with it, and it would require more time than is available for the thesis. Furthermore, it is not required for answering the question formulation of whether hash trees can improve inconsistency detection compared to a complete traversal algorithm. Therefore, this will instead be considered future work. How this might impact testing is described in section 7.2.

This project will not include writing code that performs the actual updating of the schemas after the inconsistencies have been detected, it will only implement the building of the hash trees and the hash tree inconsistency detection algorithm. The reason for this decision is simply as some code already exist for performing the updates. Furthermore, the purpose was to investigate if the inconsistency detection algorithm’s performance can be improved, and the updating routine is therefore out of scope.

1.9. Outline

Chapter 2 presents the background for the project, including information on Merkle trees and some systems used by Cisco that are relevant for this work, as well as presenting related work, in order to give the reader all information required to understand the project work. Chapter 3 describes the research process and research paradigm used in this work, as well as other information that can be used to replicate the results. Chapter 4 presents the system architecture of the implemented prototype. Chapter 5 presents the result of the work, including an analysis and discussion of the results. Chapter 6 presents conclusions, limitations and future work.
Chapter 2

Background

This section presents the necessary background information needed to get an understanding of the project work. This includes a background on internal Cisco systems used as well as a description of hash functions, Merkle Trees, Merkle proofs and anti-entropy repair in more detail. Finally, a section describing related work done in the field is included. The section ends with a summary.

2.1. Relevant Systems

In this section, some of the systems developed and/or used by Cisco is presented. This includes Cisco’s NSO, CDB and the YANG data modeling language. Cisco as a company is introduced in chapter 1.

2.1.1. NSO and CDB

The NSO getting started guide[3] that customers get with the service describes how NSO works and can be used. It states that NSO is a tool developed by Tail-f to help with creation and configuration of network services, as this is often complex and often requires configuration changes to every device in the service chain. Changes need to be made concurrently across all devices and all configurations must be synchronized. NSO solves all this so that this work does not need to be handled manually by the customers. It acts as an interface between the configurators, which are network operators and automated systems, and the underlying devices in the network.

According to Cisco[5];

Cisco® Network Services Orchestrator (NSO) enabled by Tail-f® is an industry-leading orchestration platform for hybrid networks. It provides comprehensive lifecycle service automation to enable you to design and deliver high-quality services faster and more easily.

NSO is a network automation service that configures devices and allows customers to add, edit and delete services without disrupting the overall service[5].
NSO provides the following key functions[3]:

- Representation of the services.
- Multi-vendor device configuration modification in the native language of the network devices.
- Configuration Database (CDB) with current synchronized configurations for all devices and services in the network domain.
- Northbound interfaces that can be accessed via WebUI or with automated systems using REST, Python, NETCONF, Java or other tools.

According to Cisco’s developer page, CDB is the configuration database where all the device configurations are stored[7]. CDB contains NSO’s view of the complete network configuration. It is a tree-structured database where the schema is in YANG format (see section 2.1.2 below). All information stored inside of NSO is validated against the schema.

### 2.1.2. YANG

According to RFC 7950 that describes the YANG data modeling language, YANG is used to model for example configuration data for network management protocols[8]. A YANG model defines hierarchies of data that can be used for NETCONF operations such as configuration. A YANG schema provides a description of data sent between a NETCONF client and server. The data is modeled as a tree where each node has a name and either a value, in the case of a leaf, or a set of child nodes, in the case of a parent. Nodes can have different kinds, for example container, list, and leaf. A container is used to define an interior data node in the tree. It does not have a value, instead it contains a set of children. A list is also used to define an interior data node, and contains a list of for example leaves or other containers. The leaf statement is used to define a leaf node in the schema tree. This is a node that has a value but no children.

The following is an example of a small YANG schema. All examples are from RFC 7950[8].

```yaml
container system {
  container login {
    leaf message {
      type string;
      description "Message given at start of login session.";
    }
  }
}
```

YANG is encoded to either XML or JSON when making an instance of the schema. The example above would be encoded into XML as;

```xml
<system>
  <login>
    <message>Good morning</message>
  </login>
</system>
```

A value in YANG has a type, for example int32, which is a 32-bit signed integer, and allows for defining own types based on a derived type. For example, defining a type “percent” would be done as;
typedef percent {
    type uint8 {
        range "0 .. 100";
    }
}
leaf completed {
    type percent;
}

Typedef is the definition of type “percent”, with the derived type uint8, unsigned 8-bit integer. The leaf “completed” uses this type to show completeness percentage. This would be encoded in XML as:

```xml
<completed>20</completed>
```

For the remainder of this report, the example schema below will be assumed for future examples. It’s a combination of two YANG examples in RFC 7950 [8].

```yang
module system{
    namespace "http://com/example/system";
    prefix system;

    container login {
        leaf message {
            type string;
            description "Message given at start of login session.";
        }
    }
    list user {
        key "name";
        leaf name {
            type string;
        }
        leaf full-name {
            type string;
        }
        leaf class {
            type string;
        }
    }
}
```

Assuming three users named glocks, snowey and rzell, this would be encoded in XML as:
<system>
  <login>
    <message>Good morning</message>
  </login>
  <user>
    <name>glocks</name>
    <full-name>Goldie Locks</full-name>
    <class>intruder</class>
  </user>
  <user>
    <name>snowey</name>
    <full-name>Snow White</full-name>
    <class>free-loader</class>
  </user>
  <user>
    <name>rzell</name>
    <full-name>Rapun Zell</full-name>
    <class>tower</class>
  </user>
</system>

This example schema can be represented in tree form as shown in figure 1 below.

![Figure 1: YANG tree structure for example module "System"](image)

### 2.2. Merkle Trees and Hash Trees

This section presents a description of Merkle trees, how they can be used for inconsistency detection and other related information that was gathered as necessary background for understanding how to build and use hash trees with wanted properties in a way that would fulfill the requirement for the system to implement.
2.2.1. Hash functions

As described in “Network Security Essentials: Applications and Standards” by Stallings[9], a hash function is a function that takes some arbitrarily-sized data as input and maps this data to a fixed-sized output. Hash functions can be used for a number of different purposes, and will have different properties or requirements depending on that purpose. For example, a cryptographic hash function is a hash function that aims to guarantee a number of security properties. A cryptographic hash function generally has the following properties[9]:

1. The hash function can be applied to data of any size.
2. The hash function produces a fixed-sized output.
3. The hash function for any given input is relatively easy to compute.
4. It is easy to generate a hash code given some data, but virtually impossible to generate the data given the hash code.
5. Given some data, it is impossible to find an alternative data that maps to the same hash code as the original data.

There are additional properties that can be added to further strengthen the hash function.

2.2.2. Merkle Trees and Hash Trees

Merkle Trees are a form of hash tree, where data is hashed in tree form[10]. The structure of the tree is as follows; each non-leaf node in the tree is a hash of its children nodes, while the leaf nodes are hashes of data blocks[10]. Figure 2 shows an example of a Merkle Tree. Each node is named for clearness. For example, the root node is named H(ABCD).

![Figure 2: Merkle tree structure example](image_url)
As can be seen in the figure, H(A) and H(B) are leaf nodes that are hashes of the data blocks A and B respectively. They are then hashed together to form their parent H(AB), which in turn is hashed together with H(CD) to form H(ABCD), which is the root.

Because of property 5 of hash functions, that states that given some data, it is impossible to find an alternative data that maps to the same hash code as the original data[9], a Merkle tree has the property that if two hashes are compared and differ, that means that they have been hashed from different data. So for example, if block A is changed, this means that H(A) will be different, which means that H(AB) will be different, which finally means that the root will be different. This allows for some interesting use cases for Merkle trees. For example, Merkle trees can be used for verification of data, for example checking if the data is part of the tree and if it has been altered. This is called a Merkle Proof. Merkle proofs are described further in section 2.2.3. Merkle trees can also be used for inconsistency detection between different versions of replicas. This is described more in section 2.2.4.

### 2.2.3. Merkle Proof

A Merkle proof can be used to verify that a block of data belongs to a Merkle tree, as the structure of the tree makes it easy to identify where changes in the tree occur. This can be done by only checking a small subset of the hashes. This is done by reconstructing the Merkle tree from the data, and then comparing the resulting root hash to the root hash of the original tree. If they are the same, the data belongs to the original tree. If they differ, the data does not. This is a way of for example checking the integrity of data that is exchanged between two parties, which means checking if the data has been altered in some way during the exchange by an adversary. The process of using Merkle proofs to check if some data is part of the tree is described in [11].

Checking if a data block is part of the tree works by recreating the branch built on the data block[11]. For example, assume that in the tree from figure 2, we want to find out if a data block C' is part of the tree. This requires access to H(D) and H(AB). This would entail the following steps:

1. Hash C' to get H(C')
2. Hash H(C') and H(D) together to get H(C'D)
3. Hash H(AB) and H(C'D) together to get H(ABC'D)
4. If H(ABC'D) = H(ABCD), this means that H(C') = H(C), which in turn means that C' = C and C' is therefore part of the tree. If H(ABC'D) is not equal to H(ABCD), C' is not part of the tree.

### 2.2.4. Anti-Entropy Repair

Anti-entropy repair is the process of detecting inconsistencies in replicas of data. This is often done using Merkle trees, for example in Apache Cassandra, Dynamo and Riak (see section 2.3). Each replica has its own Merkle tree and is compared to all the other replica trees. If an inconsistency is detected, the data is updated. The comparison is done in the following steps:

1. Compare the root of the trees.
   a. If the roots differ, move to step 2.
   b. If the roots are the same, all data is the same between the replicas, so there are no inconsistencies. The comparison is then done.
2. Compare the left and right children of the root. If a change has occurred to the tree, at least one of them must differ between replicas.
3. Move down the branch where data differs.
4. Continue until the leaf is reached

When the leaf is reached, the data block corresponding to that leaf is differing between the two trees. The data can now be updated. If numerous data blocks have changed, their respective branches are traversed. This provides an efficient way of detecting differences in data. Inconsistencies can be detected without having to iterate through the whole tree, only the branch(es) that are inconsistent. If all data blocks have changed, all branches of the tree have to be traversed, and all data blocks must updated, which means no time is saved compared to checking all data blocks linearly. However, this is the worst case scenario, and depending on what the inconsistency detection is used for, this is likely an improbable scenario. This process of using anti-entropy repair with Merkle trees is described in [12].

For example, assume the tree from figure 2, which is the first replica of some data, and another tree, which is the section replica, where data block C has been changed to C'. Therefore, C is now the old version of the data and needs to be updated. The inconsistency detection between the replicas of the data proceeds as follows;

1. Compare the root hashes, H(ABCD) and H(ABC'D). They are different, so there is some inconsistency between the replicas.
2. Compare H(AB) for both trees with each other. They are the same, so there are no inconsistencies in the left subtree.
3. Compare H(CD) with H(C'D). These are different, which means there is some inconsistency in the right subtree.
4. Compare H(C) and H(C'). These are different. This is a leaf node, so one inconsistency is found with data block C between the replicas.
5. Compare H(D) for both trees. They are the same, so both trees have the same version of data block D.
6. Perform an update procedure to synchronize the replicas, in this case updating C to C', which is the newer version.

2.3. Related Work

This section presents previous works related to this project. All the presented systems use Merkle trees for detecting inconsistencies between different replicas in replicated stores.

2.3.1. Dynamo

The original paper on Dynamo describes Dynamo as a highly-available and scalable distributed key-value storage system[13]. Dynamo was developed by Amazon to provide an “always-on” experience for its customers. To do this, Dynamo sacrifices consistency for availability. Therefore, Dynamo has eventual consistency, which means that all updates reach each replica eventually.

The paper further states that Dynamo is used to manage the state of services that have very high reliability requirements, as the goal is to provide a service that is always available. In Dynamo, data is partitioned and replicated using consistent hashing. Data consistency is facilitated by versioning and maintained by a quorum-like technique, and Merkle trees are used as the synchronization protocol. Dynamo uses a gossip-based failure detection and membership protocol, where nodes can be added and removed without any manual involvement.
Dynamo uses Merkle trees for inconsistency detection between replicas of data[13]. Each node in Dynamo has a key range, which is the set of keys covered by a virtual node. Each key range is represented by a Merkle tree. When checking if keys in the key range are up-to-date, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they have in common. Anti-entropy repair is then used as described in section 2.4 to detect any differences between the key ranges of the nodes.

The paper described Merkle trees for inconsistency detection as fast and with the advantage of minimizing the amount of transferred data, as data only needs to be transferred between nodes if the root hash differs between them. The principle advantage that is mentioned, however, is that each branch of the tree can be checked independently without the nodes having to download the entire tree.

### 2.3.2. Apache Cassandra

The original paper on Cassandra describes Cassandra as a distributed, scalable, highly available storage system created by Facebook and designed for managing large amounts of data[14]. According to the paper, Cassandra manages the persistent state even during failures which makes it highly reliable. Cassandra is a membership protocol that manages, among other things, partitioning, replication, failure handling and recovery.

Just Like Dynamo, Cassandra uses Merkle trees for replica synchronization[12]. DataStax provides official documentation for Cassandra and describes how anti-entropy repair is used by Cassandra[12]. Anti-entropy repair is used for routine maintenance of replicas. Inconsistencies can occur when nodes fail or data is changed or deleted, which is why the repair procedure needs to be run to keep the replicas synchronized. The procedure works as described in section 2.4. The node that initiates the repair becomes the coordinator of the operation. When building the Merkle trees, this node has the responsibility to determine peer nodes with matching ranges of data. The peer nodes then in turn trigger a validation compaction, which reads each row and determines a hash for it, before storing the result in a Merkle tree. The peer node then returns the Merkle tree created to the coordinator node.

Cassandra has two types of repair; full repair and incremental repair. Full repair creates a full Merkle tree and compares the data against data on other nodes. Incremental repair only builds Merkle trees for data that has not been repaired previously, which reduces the time it takes to repair new data[15]. This was introduced in Cassandra 2.1 as it is expensive to calculate a new tree each time data is repaired[15].

The repairs can also be sequential or parallel. Sequential repair takes action on one node after another. Parallel repair repairs all nodes with the same replica data at the same time. Parallel repairs is a faster operations and is therefore used to save time[15].

### 2.3.3. Riak

Riak is a distributed, highly available, scalable, key-value storage system created by Basho Technologies[16]. As Dynamo and Cassandra, Riak partitions data over a set of nodes in a cluster built as a ring, with no single point of failure, that sacrifices consistency for availability.

According to the Riak documentation[17], Riak uses two different methods for synchronizing replicas: read repair, also called passive anti-entropy repair, and active anti-entropy repair. Read repair is used in versions prior to 1.3. It handles object conflicts only when a read
request reaches Riak from a client. In read repair, the node coordinating the read request is responsible for detecting inconsistencies among nodes, and will start the repair process if an inconsistency is detected.

Active anti-entropy enables conflict resolution to run as a continuous background process. It is useful for data that is not read for long periods of time. This kind of data is not reachable by read repair, and active anti-entropy is then needed for handling such situations[17].

According to the documentation, Riak, like Dynamo and Cassandra, uses Merkle trees for detecting inconsistencies between replicas of data. However, Riak uses persistent, on-disk hash trees instead of in-memory hash trees. This allows Riak to run anti-entropy repair with minimal impact on memory usage and to restart nodes without needing to rebuild the Merkle trees. The Merkle trees are updated in real time, which reduces the time to detect and repair inconsistencies in data.

2.4. Summary

Merkle trees are hash trees where each node in the tree is a hash of its child nodes, and the leaf nodes are hashes of data blocks. Merkle trees can be used for verification of data, which includes integrity checks and finding inconsistencies in the tree or between trees. This can be done using Merkle proofs, where a new tree is constructed from some data, and this tree’s root is compared to the original Merkle tree root. If the roots differ, the data is not part of the original tree.

A technique called anti-entropy repair can be used to detect inconsistencies between different replicas of data. One Merkle tree is built for each replica, and to detect inconsistencies all trees are compared to all other trees. If differences are detected in the tree, the differing replica(s) is/are updated. The comparison starts at the root and moves down the tree, following the branch where differences are detected until the differing data is reached. This data is then updated. If there is no difference found in the root, the replicas are identical.

This method of inconsistency detection in replicas are used by Dynamo, Apache Cassandra and Riak.
Chapter 3

Methodology

This section describes the research process and research paradigm for the project, as well as what experiments and measurements were done and on what systems and hardware in order to make the results replicable. The research process describes the steps taken for solving the problem, while the research paradigm describes the methodology used for the project, which in this case is the engineering design process.

3.1. Research Process

The research process for this project involved creating a prototype of the system to be implemented and testing this prototype in comparison to another prototype of the complete traversal algorithm used by Cisco, in order to answer the question formulation. An alternative strategy would be to perform an entirely theoretical work based on existing data on Merkle trees and hash trees used as inconsistency detection algorithms, and use these measurements to make an estimated guess on how much hash trees would improve the performance of the current inconsistency detection system used by Cisco. However, for this project it was opted to instead implement a prototype of the suggested new hash tree algorithm in order to get a more accurate estimation of the improvement in performance. This will just be an estimation, as the code will not be integrated with the existing Cisco system, which means that it cannot be known exactly how much time the new algorithm saves when it is used in real-life scenarios. It will however, give a more accurate estimation than a theoretical approach as a solution of the algorithm is implemented and can be tested for performance.

The process of developing the prototype was inspired by the engineering design process described in section 3.2 below, and consisted of information gathering, implementation and testing. The whole research process followed the below steps;

1. Problem Formulation/identify the problem: the first step was to formulate the problem that was to be solved and the question that needs to be answered to solve it, in order to understand what needed to be done to solve the problem. The problem formulation for this project is described in section 1.2.
2. **Idea for solution**: when the problem had been identified, the next step was to determine a possible solution to the problem that could be developed and tested in this work.

3. **Literature study**: In order to develop a prototype to be tested as a possible solution, sufficient knowledge needs to be attained. In this case, the literature study was divided into three information gathering parts; background on Merkle and hash trees, background on internal Cisco systems like NSO and CDB, and background on the engineering design process. The result of the literature study for the first two phases can be find in chapter 2: Background. The result from the third phase can be find in the next section, section 3.2, where the engineering design process is described.

4. **Understanding the current system used by Cisco**: when enough information had been gathered to understand how the internal systems in Cisco are used, how they would have to be used in this project, and how Merkle trees and hash trees are built and can be used for inconsistency detection, the next step was to scrutinize the complete traversal algorithm currently used by Cisco in order to understand how the inconsistency detection is done now, to understand its limitations and what result the implemented prototype would have to return. This gives an understanding of what requirements the new system must fulfill.

5. **Implementation of prototypes**: this step is where the prototypes developed for answering the question formulation are implemented. This includes implementation of the hashing, how to build the hash tree and how to perform the inconsistency detection. This is described in more detail in chapter 4.

6. **Determining average number of nodes that need to be traversed**: as the implemented hash tree would only need to traverse branches that contains data that have been modified between versions, the number of modifications is one factor that determines the performance of the implemented system. In addition, the branch only needs to be traversed until the inconsistency is reached (see chapter 4), so the positioning of the node that has been changed in the tree is also a factor that affects the performance. Therefore, it is important to determine the best and worst case scenarios with respect to changes, but also an average scenario as this gives a more accurate representation of how the system can be expected to perform in real-world scenarios. Therefore, the average number of nodes that have to be traversed between versions of schemas need to be determined for the hash tree and the complete traversal approach algorithm used by Cisco.

7. **Performance testing**: in this phase the performance of the complete traversal algorithm compared to the hash tree algorithm developed in this project is tested in order to determine by what factor, if any, the performance is improved by a hash tree approach. The performance testing will be done for different size schemas and best, worst and typical case performance will be determined. Typical case performance will be determined by testing the performance of several real-world update scenarios for each schema. More on how this is done is described in section 3.3.

### 3.2. Research Paradigm

When developing the prototypes for this project, the engineering design process methodology was used as inspiration. The engineering design process is a process followed by engineers when designing a product in order to solve some problem. According to ABET,
“Engineering design is the process of devising a system, component, or process to meet desired needs”[18]. In their book on engineering design, Dym and Little state that[19]:

Engineering design is a systematic, intelligent process in which designers generate, evaluate and specify designs for devices, systems or processes whose form(s) and function(s) achieve clients' objectives and users' needs while satisfying a specified set of constraints.

The engineering design process follows a series of steps. The process is often iterative, which means that some steps can be repeated several times to meet the desired results[18]. However, which steps should be included in the process varies between definitions and authors[19].

Tayal, in his book “Engineering Design Process”, describes the engineering design process as a “formulation of a plan or scheme to assist an engineer in creating a product”, and describes it as finding a problem, identify possible solutions and implement a chosen solution[20]. He suggest the below steps for the process, although he states that the steps are only to be used as guidelines and do not have to be followed exactly[20].

1. **Define the problem**: this phase includes identifying the problem, what is to be accomplished, project requirements and limitations, and goals.
2. **Do background research**: the purpose of this phase is to gather the necessary information to carry out the project. Consideration should be given to previous work and solutions.
3. **Specify requirements**: here the requirements on the system or product developed are identified. This could for example be requirements on the hardware and/or software, availability and testability.
4. **Create alternative solutions**: this phase involves coming up with possible solutions to the problem and assessing them.
5. **Choose the best solution**: in this phase the most promising solution is chosen.
6. **Do development work**: this includes designing and modeling the product and generally planning what the product should look like.
7. **Build a prototype**: a working product is built. This is just a prototype as the process can be iterated several times to improve the product.
8. **Test and redesign**: test the product and perform necessary adjustments.

Ertas and Jones, who have also authored a book on engineering design, state that the design process begins with an identified need and concludes when the product has been tested and deemed satisfactory[21]. They present the below steps in the process. These steps are, according to Ertas and Jones, generally applicable, but individual projects might require variations and skipping of steps. He states that this is especially true for smaller projects.

1. Recognition of a need
2. Conceptualization and creativity
3. Feasibility assessment
4. Establishing the design requirements
5. Synthesis and analysis in the design process
6. The organization/work breakdown structure
7. Preliminary design
8. Detailed design
9. Production process planning and tooling design
10. Production
11. The product realization process
12. Design for manufacture and assembly

As stated by Ertas and Jones, it is clear from the steps that they are developed for work with bigger projects for larger companies. However, the principle of identifying a need, finding a solution, designing the product and developing the product is still present, though Ertas and Jones do not include testing in the development process.

### 3.3. Planned Measurements

Merkle trees and the hash tree structure implemented for this project have the property that more changes lead to more branches having to be traversed, as all branches that contain changes lead to some change in data that has to be detected. This means that the performance of the algorithm depends on how many changes has been made to the data. In the case if this work, the performance also depends on where the change lays in the tree. This is because each node contains data, and if there is no change in the hash of the combined children of a node, no data has been changed in that node’s subtree. The structure of the hash tree is described in detail in section 4.2.

Consider for example a tree where only one change has been made. If this change is made to a node near the root, the whole branch does not need to be traversed, only down to the differing node. If the differing node is a leaf, the whole branch needs to be traversed as the change is made to the bottom of the tree. There can also be deeper or more shallow branches in the tree, so a change to a leaf in a shallow branch will not affect performance as much as a change to a leaf in a deeper branch. This also means that the number of changes can have a smaller effect if they all occur in the same branch. If two changes have been made to the same branch, only this branch needs to be traversed, but if these two changes are made to different branches, both these branches must be traversed. Similarly, because all children of a node needs to be traversed to determine which child has changed, two changes made to two different siblings will not affect performance much more than only one change among the siblings, as long as there are no changes to the subtrees if the siblings. In this case too, more changes can affect performance less if they are done to siblings than fewer changes spread out in the tree. Therefore, the positioning of the changes, in addition to the number of changes, is relevant to the performance of the algorithm. It can therefore be concluded that the performance depends on the number of nodes that has to be traversed, which in turn depends on the number and positioning of the nodes where changes have occurred.

In order to determine how the hash tree algorithm would perform in different situations, tests must be done to show the level of improvement in the best and worst case. It is also important to determine how the algorithm performs in real-life situations, as this will give an overview of how it would perform when the program is used in real scenarios that will occur in the system. This would give a typical case performance. Therefore, schemas will be tested with specially constructed best and worst cases as well as real-life cases of updates that have been made to the schemas.

The same schemas will be used for special case testing and real-life scenario testing. The schemas will be of varying sizes to see how this factor effects performance. For each schema the following special cases will be tested:

- All nodes have changed. This will give a worst case performance estimate.
- No changes have been made. This gives a best case performance estimate.
- A single change has been made to a leaf node near the root.
- A single change has been made to a leaf node in one of the deepest branches.
The last two cases listed are included to see how the positioning of the change affects the performance. In theory, as mentioned above, it can be assumed that a change to a leaf node near the root, which is a change to a shallow branch, will mean less nodes traversed than if a change has been made to a leaf node in a deeper branch, and therefore the algorithm should finish faster.

The real-life examples will be used not only to determine the average performance of the system, but also the average number of nodes that need to be traversed for the hash tree, as this impacts the performance. As mentioned, this depends on the number of differences as well as the positioning of the differing nodes.

All testing will be done on six schemas of different sizes to determine how the size affects performance. For each schema version update the inconsistency detection algorithms will be run 50 times. The time for the inconsistency detection will be reported for each run, and the median will then be calculated for each schema. The median value is used instead of the average, as the average value is affected by possible spikes occurring for some runs. The median will be used to determine the factor of speedup, if any, the hash-tree inconsistency detection will yield for differing sizes of schemas. From this, it can be determined if in fact hash trees can be used to improve the performance of database schema inconsistency detection. The result of the performance testing is presented in chapter 5.

When a schema enters the system, a hash tree must be built for the schema in order for the hash tree comparison to take place (see chapter 4 for details). When a schema is updated at the customer, the process is divided into compile time and runtime. The hash tree is built during the compile time. This time is relatively short and does not directly affect the customer, as it is done on a Cisco server. The runtime is the time from when the customer performs a load command to when the new schema version is saved to the CDB database. This is where the inconsistency detection takes place and is the time that affects the customer. This is also the time that Cisco wants to reduce. Therefore, the time it takes to build the hash tree, which is done during the compile time, is not interesting for Cisco. Consequently, no performance tests will be done measuring the time it takes to build the hash tree, as this is not relevant for Cisco’s purposes. Only tests measuring the inconsistency detection time will be performed.

3.3.1. Test Environment

All tests will be performed on a 64 bit Lenovo P51s Thinkpad with 4 cores, 16GB RAM and Intel Core i7-7600U CPU processor with 2.80GHz processor speed. The operating system used is Ubuntu 18.04. The prototypes were written in the Erlang programming language with Erlang/OTP version 20.
Chapter 4

System Architecture

The system built consist of two separate components that can be used individually and are run at different times. These are the component that builds the hash tree from an existing YANG tree, and the component that performs the inconsistency detection given two hash trees. A third component exist for performance testing purposes, which traverses all elements in a YANG schema to simulate the current algorithm for inconsistency detection used by Cisco.

In order for the inconsistency detection to take place, a hash tree must first be built from the YANG schema. Each version of a schema will be represented by its own hash tree. When a new version of a schema enters the system, which is when a schema change has been triggered by a device configuration update, a hash tree is built for this schema. This schema will be compared to the old version of the schema and the necessary updates are made. The result from this is then stored in persistent storage and used when yet another new version of the schema enters the system.

The below sections presents a description of how the hashing was done, how the hash tree is built and how the inconsistency detection was implemented, as well as how the performance testing was performed.

4.1. Hashing

The process of hashing the tree includes deciding on a hash function to use for hashing the node data of the tree, deciding which fields of a YANG component to hash, and how to append the hashes of a node’s children with each other in order to form the children hash of a node.

4.1.1. Choice of Hash Function

When selecting an appropriate hash function for the purpose of this project, the hash function properties presented in section 2.2.1. were considered, in order to determine which properties the chosen hash function needs to uphold.
Property 1 and 2 that state that the hash function can be applied to data of any size and produces a fixed-sized output respectively are basic properties of a hash function. They need to be upheld for the purposes of this project, but it can also be assumed that any hash function will uphold them as it would not be a hash function otherwise.

In the case of the project, the purpose of the hash algorithm is to give a unique fingerprint of some data. This is crucial as it is the difference of two hashes for some data that determines if the data needs to be changed. What could happen in the worst case is that two differing versions of the same node in two different trees will have the same hash. This would lead to the algorithm wrongly determining that the node has not changed between versions, which leads to an inconsistency being missed. This would mean that an incorrect version of a schema would be stored to persistent memory. It is therefore desirable that the hash algorithm fulfills property 4, that different data cannot map to the same hash. However, it is not important if two nodes in the same schema have the same hash, as nodes in one schema will never be compared to each other, or that nodes for different schema versions with differing names have the same hash, as a node will only be compared to another node with the same name. Two nodes with different names but the same hashes will never be compared to each other, so this will not yield a fault.

It is not important that the data cannot be obtained from the hash, as the hash tree does not contain sensitive information and will only be used in Cisco’s internal system. It is not a concern whether it is possible to get the data from the hash tree. Therefore, the hash function chosen for this project does not necessarily need to uphold property 5, so this will not be considered when choosing the hash function.

A factor that is important to consider instead is how fast the hash can be calculated, which relates to property 3, and how many bits the hash consists of. How fast the algorithm can be calculated affects the time it takes to build the hash tree, and the length of the bits determine how long it will take to compare the hashes of the schemas. As mentioned before, the time to build the hash tree does not affect the customer, as this is done at compile time. However, it is still preferable to have this be performed as fast as possible as it would save time for the Cisco system, even though it does not directly affect the customers. The length of the hash however does affect the speed of the inconsistency detection. Two hashes of length 128 bits can in theory be compared to each other twice as fast as two hashes of length 256 bits, as the hash is only half the size. Therefore, it is desirable to choose a hash algorithm that can be computed fast and in addition is relatively short.

Erlang’s crypto library supports the SHA-1, SHA-2 and SHA-3 hash function families, MD4, MD5 and RIPEMD-160[22]. SHA-1 and RIPEMD-160 are 160 bit hashes, SHA-2 and SHA-3 can be 224, 256, 384 or 512 bits, MD4 and MD5 are 128 bits. From this it can right away be determined that MD4 and MD5 would be the fastest choices for the inconsistency detection.

From experiments done on this subject by the cryptographic library Crypto+++, it has been determined that MD5 is the fastest to compute, with MD4 not being present in the experiments[23]. Both RIPEMD-160 and SHA-256 perform poorly[23]. According to MD5’s RFC, MD5 is slower than MD4[24]. From this, the conclusion can be drawn that MD4 is the fastest of the available hash functions in Erlang. This means that MD4 would be the most appropriate choice for both building the hash tree and performing the inconsistency detection when considering time performance.

In 2005 it was found that two messages can deliver the same SHA-1 hash using fewer than $2^{90}$ operations[25]. This means that SHA-1 does not uphold property 4 of a hash function.
This property is also called collision resistance[24]. MD4 and MD5 have also been shown to not be collision resistant[26, 27]. However, according to MD4’s RFC, it is conjectured that it is "computationally infeasible to produce two messages having the same message digest"[28]. More specifically, the RFC states that it is conjectured that the probability of two different data giving the same hash is on the order of $2^{64}$ operations. This RFC is dated by now, but the fact remains that even if collision attacks are possible, the risk of two different data accidently mapping to the same hash are incredibly low and it is infeasible that this should happen with schemas containing at most a few hundred thousand nodes. Therefore, even though MD4 technically does not uphold property 4, it will still be considered the best choice for the purposes of this project, due to the low risk of accidental collisions and the computational speed.

4.1.2. Determining What Data to Include in the Hash

A YANG component in a YANG schema contains some information about the component. This information is what a YANG node in the YANG tree representation consist of. This includes information about the node’s name, type, status and children. For example, in the YANG module presented in section 2.1.3, there is one node for the container “Login”, which contains information such as the component name (“Login”), the component kind (“container”), the module it belongs to (“System”), the component’s children (“Message”), and so on. Overall the nodes in the YANG tree contain 20 fields of data. For some of these fields it is irrelevant whether or not they have changed. These fields do not need to be included in the hash, as it should not affect whether or not the node needs to be updated. For example, it is unnecessary to hash the “name” and “children” fields, as if a node changes name it is considered a new node (see section 4.3) and if any children has been added or removed this will be detected when checking the combined hash of the node’s children.

4.1.3. Combining Hashes

To form the children hash of a node, all the hashes of the node’s children must be combined into one hash. One way of doing this would be to append the hashes of the children together and then hash the result. However, this operation is not commutative, which would mean that if the children nodes are reordered among themselves, the hash will differ. For the purposes of this work, the ordering of sibling nodes do not matter. Two schemas with the same structure and the same nodes where the sibling nodes have different ordering are considered the same schema, so no update procedure needs to take place. As the inconsistency detection algorithm matches the names of the nodes and compares them, this way of appending the hashes would not lead to a faulty inconsistency being detected (see section 4.3). However, it would lead to unnecessary traversal of the tree, as the children hashes would differ down to the nodes where the order has changed, and these branches would be unnecessarily traversed. Therefore, another method of combining the hashes needs to be used for optimization purposes.

An alternative way to combine the hashes is to XOR them together. Bitwise XOR is a commutative operation, so the order does not affect the result when XORing data. Therefore, the order of the children can change without the resulting children hash of a node being affected. The algorithm first XORs the first two children together, then XORs the result of this with the next child, and continues like this for all children of a node. The result is used as the children hash of that node.
4.2. Hash Tree Structure

The data that is to be hashed arrives to the system in tree form, as YANG schemas are structured as trees. Elements in the YANG schema are represented as nodes in the Erlang representation of the YANG tree. All components are subject to change and those changes need to be detected. The question then to consider is how to structure this data into a hash tree. A true Merkle tree has a binary structure built on a linear array of data blocks. One alternative of hashing the YANG tree would be to serialize the tree data into a linear array and build a Merkle tree from it. However, this would require some system that keeps track of the original YANG tree structure of the nodes, so that the nodes that have been changed can then be identified and updated in the original YANG tree. Another alternative would be to build the hash tree to preserve the structure of the YANG tree. This has the advantage of making it straightforward to preserve the path to the nodes that have been changed, so that the original tree can then easily be traversed to find the differing nodes. Therefore, it was decided that this approach would be used. This then means that each node in the original YANG tree will be hashed to detect if its data hash has changed in the hash tree.

An important aspect then is to consider how to preserve the properties of a Merkle tree while also preserving the YANG structure. The basis of inconsistency detection, or anti-entropy repair, in Merkle trees is that if the hash of a node has changed, at least one of the child nodes has changed. This is used to traverse the tree to find which data block(s) has/have changed. If this property is not preserved, all branches must be traversed no matter if a change has actually been made to the data blocks, which would defeat the purpose of building a hash tree in the first place. Therefore, this property needs to be preserved when building the hash tree, as otherwise the complexity would remain linear. This means that, in addition to comparing the node data to see if the current node has changed, the node also needs to contain hashes of the children, so that it can be determined if a change has occurred further down the tree, thereby determining if the subtree of the node needs to be traversed.

A node in the hash tree will therefore contain the following data:

- Hash of the YANG data (for example name, kind and type) to detect changes in that data, hereafter referred to as “data hash”.
- Combined hash of all children hashes to detect if a change has occurred to the node’s subtree, thereby determining whether or not the child nodes need to be traversed. This will be referred to as “children hash”.
- Combination of the data hash and a children hash. This hash will be checked first to determine if any change has occurred at all to the node or the subtree. This will be referred to as “node hash”.

The node hash is necessary as when checking the hash of a child node, it must differ both if the data hash has changed and if the children hash has changed. Therefore, this is the hash of a node that will be combined with its siblings’ node hashes to form the children hash of the parent. If the children hash of a node would consist of the data hashes of its children, this would only detect if a change has been made to the data of one or more children of the node. It would contain no information about changes further down the tree. So if the children have not changed, but the children’s children have changed, the corresponding branches would not be traversed and the changes would not be detected, as there would be no change to the children hash of the node, only to the children’s children hashes. In contrast, if the children hashes are combined to form the children hash of the parent, this would never capture any data change. This is because the leaves have no children and therefore empty children hashes, so the parents of the leaves would then be a combination of empty hashes and this would continue up the tree, thereby not capturing any change. Therefore, using the node
hash, as a combination of the data hash and children hash, for each child when combining them into the children hash of the node, will allow the children hash to show if any change at all has happened to any node in the current node’s subtree.

So for example, in figure 3 below, the node “Name” would have an empty children hash as it is a leaf, a data hash that shows if any change has been made to the node itself, and a node hash that is a combination of the data hash and children hash, which in this case will be the same as the data hash as the children hash is empty. The node “User” will have a children hash that is a combination of its children’s node hashes, which captures if any change has occurred to the children. The data hash will be a hash of the original YANG node data for “User”. The node hash will be a combination of the data hash and children hash, thereby showing if any change has occurred to either hash. The same is true for node “System”, so its children hash will show if a change has occurred to the node hashes of either “Login” or “User”, which in turn shows if a change has occurred to either the nodes themselves or their children. This way, the node hash of a node X shows if any change at all has occurred to X or any node in X’s subtree.

The data hash and children hash are not necessary to include in the hash tree nodes, it would be enough to have the node hash as a combination of them. When the node hash differ, one would then check the node’s data fields to see if any data for the node differ. If not, a change must have been made to the node hash of one or more children. If one or more data fields differ, the children must be traversed in any case as it is possible that one or more children differ too. Using this method would take more time, partly because the data fields take longer to check than a 128-bit data hash, and partly because the children must be traversed if the node hash has changed even if no change has occurred further down the tree. It would then be detected for each child that no change to the child’s node hash has been made, but this requires that all children are traversed unnecessarily. The data hash and children hash are therefore included in the hash tree node representation for optimization purposes, as it can save considerable time.

Figure 3: Example of hash tree structure
How the hash tree will be traversed to detect inconsistencies is described in section 4.3. Figure 3 above shows the structure and contents of the hash tree based on the YANG schema example from section 2.1.2.

### 4.2.1. Implementation

When building the hash tree, the existing YANG tree is traversed down to the leaves, and then converted node by node to a hash tree from the bottom up. When reaching the leaves of a branch, the leaves are hashed in order. As a leaf has no children, the node hash and data hash will be the same hash, and the children hash is empty. The traversal will then reach the parent node of the leaves, and hash it. The children hash is created as described in section 4.1. The traversal will continue up the tree until the root is reached. When traversing the example tree from figure 3, the nodes will be built in the following order; “Message”, “Login”, “Name”, “Full-Name”, “Class”, “User”, “System”. The traversal starts with the left branch, building the tree upwards, and when reaching “Login”, the right branch is traversed to reach “User”, so that “User” and “Login” together can then be combined to create “System”.

As this application that builds the hash tree and the inconsistency detection application are separate applications that are run in different parts of the system, as described previously in this section, this application needs somewhere to save the output, which is the hash tree, so that the inconsistency detection application can access the trees. For now, as the product will not yet be integrated to the Cisco system, it is practical to store the result in a file. This file can then be read by the inconsistency detection application and converted back to Erlang terms. This way, the inconsistency detection application will get access to the hash tree created by the application that builds the hash tree.

### 4.3. Inconsistency Detection

As described in section 4.2 above, when traversing the tree, the node hash shows if any change at all has occurred to the current node or subtree, the data hash shows if any change has occurred at the current node, and the children hash shows if any change has occurred to any of the current node’s children’s node hashes, and therefore the node’s subtree. Therefore, all three hashes need to be compared for changes. The inconsistency detection will start at the root and follow the below steps:

1. Check node hash. If this has changed, either the node itself, the node data, has changed, or the node’s subtree has changed; proceed to step 2. If the node hash has not changed, there are no changes made to this node or its subtree. If this is the root, there is no difference at all in the tree. Otherwise, continue checking the node’s siblings, if it has any, thereby starting step one again for the next sibling.
2. Check data hash. If this has changed, an inconsistency has been found and should be reported. Proceed to step 3 in any case.
3. Check children hash. If this has changed, check each child to find where the inconsistency lay by checking their hashes again from step 1. Otherwise, this subtree does not have to be traversed further, the change was at the current node.

To make this clearer, the list below presents the steps that are taken for the example tree given in section 4.2 above. In this hypothetical case, assume node “Login”, leaf “Message” and node “User” have changed in some way. The inconsistency detection would proceed as follows:

1. Check root “System”.
   a. Check node hash. It has changed. Node data and/or children have changed.
b. Check data hash. It has not changed. This means that this node’s data is the same in both versions.

c. Check children hash. It has changed. One or more of the children’s node hashes differ, which means there is some change in the rest of the tree. All children must therefore be checked for inconsistencies.

2. Check node “Login”

a. Check node hash. It has changed. Either node data or children have changed.

b. Check data hash. It too has changed. An inconsistency has been detected and should be handled.

c. Check children hash. It has also changed. Proceed to check children.

3. Check node “Message”

a. Check node hash. It has changed. This is a leaf, so no children have changed, so the change must be in the node data. However, the algorithm will proceed to check data hash and children hash anyway. Handle the inconsistency.

4. Check node “User”.

a. Check node hash. It has changed. Either node data or children have changed.

b. Check data hash. It has changed. Handle inconsistency.

c. Check children hash. It has not changed, which means that the children’s hashes do not need to be checked. All inconsistencies in the tree have been found.

In this case, three changes were made to the tree, and four of the seven nodes had to be traversed. If one of the children to node “User” had changed instead of “Login”, there would be the same number of changes, but the whole tree would have to be traversed in order to determine which of the children to “User” had changed. This demonstrates how the positioning of the changes in the tree affects performance and not only the number of changes. If for example “User” hadn’t changed but one of its children and the node “Message” had changed, there would only be two changes to the tree, but the whole tree would still have to be traversed. So even with less number of changes, the positioning can lead to more nodes having to be traversed.

In step 2, the node hash has changed but not the data hash. This means that the children hash must differ for this node, as either data hash or children hash must have changed for the node hash to change. Therefore, sub-step c is unnecessary. This could be updated for further optimization, but it is assumed that this will not make a signification change in performance.

4.3.1. Implementation

When performing the inconsistency detection, the algorithm cannot simply check the children from left to right or right to left for both subtrees and compare the nodes that appear in the same position in both trees, as nodes can be deleted or added. For example, if in the YANG schema from figure 1 a new element called “ID” was added as the first element to list “User”, such an algorithm would compare “Name” in the first version to “ID” in the second version, then “Full-Name” in the first version to “Name” in the second version and finally “Class” in the first version to “Full-Name” in the second version. This would compare the incorrect nodes to each other and therefore report inconsistencies where there are none, instead of reporting that a node has been added to the second version. Furthermore, for Cisco’s purposes it does not matter if sibling nodes change places in the YANG schema, so if for example “Full-Name” and “Name” would change places, this is still considered the same tree and no update needs to be made.
To handle the above cases the algorithm needs a way to determine what nodes in the different schemas to compare to each other. For this, the node’s name is used to determine if it is the same node in the two different versions. If a node changes name, it will be considered a new node, as there is no way for the algorithm to determine if for example node “Name” and node “User-Name” from two different versions of a schema are the same node that has changed name or if “Name” has been removed and “User-Name” has been added without connection to each other. There are no unique fields to a node in a YANG schema that can be used to identify if it is the same node in two different versions. This means that when traversing the trees and getting to the case where a node’s children should be compared to each other, the algorithm will take the first child of the schema version 1 list of children, and check the list of children from the version 2 schema for a matching name. If a match is found, the nodes can be compared by checking the node’s hashes as described above. If no match is found, it means that version 1 of the schema has a child that does not exist in version 2 of the schema, so this node must have been removed from version 2. If all children from version 1 have been traversed but there still exist one or more nodes in the version 2 list of nodes, version 2 of the schema contains nodes that do not exist in version 1, so one or more nodes have been added to the version 2 schema.

To make this process clearer, consider the following example where the tree from figure 3 is version 1 of the schema, and version 2 have the node “Class” renamed to node “Group”. As there is no way for the algorithm to know if “Class” and “Group” are the same nodes with a name change, as they contain no unique identifier, the algorithm has to assume that “Class” has been removed and “Group” has been added. When the algorithm reaches the children of “User”, it will proceed with the following steps:

1. Start with “Name” from version 1 and see if a match is found among the children of “User” in version 2. In this case, a match is found, so continue by comparing the node hashes of the node from both versions to each other to determine if the node has changed.
2. Proceed with “Full-Name” from version 1 and see if a match is found in version 2. In this case too, a match is found, so the algorithm proceeds to compare the hashes of the nodes.
3. Proceed with “Class” from version 1 and try to find a match in version 2. In this case, no match is found, so version 2 contains no children to “User” that are called “Class”. Therefore, this node must have been deleted between versions.
4. Node “Group” from version 2 is left over. This means that a node exist in version 2 that does not exist in version 1, so this node has been added between versions.

When the trees have been traversed, the algorithm returns a tree containing all nodes that have been changed in some way. Each node in this tree has information on the node name, if the node itself has changed (or if it is just a node in the path to a node that has changed), and the node’s children that have changed, been added or been deleted compared to the later version of the schema. The changed nodes are then nodes where some other field than the name has changed, as a name change is treated as a new node being added and an old node being deleted. From this return value, it is possible to find the nodes that require an update procedure and perform the update on these nodes only.

To make this tree structure more clear, consider the schema from figure 1. This is version 1 of the schema. Assume that in version 2 node “Message” has changed, node “Full-Name” has been removed and a node “ID” added as a child to “User”. The root of the tree of changes will be node “System”, and this node has been marked as not changed. The root has two children that contain some changes either in themselves or to their children, “Login” and “User”. No children that have been added to or deleted from the root. Login is marked as not changed,
and has one child that has been changed and is marked as such, “Message”, and no children added or deleted. “Message” has no children at all. Node “User” has no children that has changed, one child that has been added, “ID”, and one child that has been deleted, “Full-Name”. This tree of changes, that is returned from the inconsistency detection of version 1 and 2 of the YANG schema from figure 1, is represented in figure 4 below.

![Figure 4: Tree of changes returned from inconsistency detection algorithm](image)

### 4.4. Performance Testing

In the current system used by Cisco, when an inconsistency is detected the updating procedure to update the schema in the database is run immediately. In addition, the algorithm is integrated in a larger system, which makes it hard to separate it from other functionality. Therefore, it would be complicated to measure only the time it takes to perform the inconsistency detection separate from the rest of the system, as this is tightly integrated. To tackle this, a separate algorithm will be implemented and tested compared to the inconsistency detection algorithm developed for this thesis. This algorithm will go through all nodes in the YANG tree and compare a node’s fields one by one, as is done in the current system. The performance of this algorithm can then be measured compared to the performance of the hash tree inconsistency detection algorithm, which will determine with what factor the time decreases, if at all. This could then be used to estimate how much time will be saved for the current system. The drawback is that this will not give completely accurate numbers, only an estimation. However, before the system developed for this project can be fully integrated with the current system at Cisco, this will yield the most accurate result. These numbers will also show in the general case how much time can be saved with hash trees for inconsistency detection compared to a complete traversal solution.
4.4.1. Implementation

The algorithm implemented for the performance testing traverses all nodes in the two versions of a YANG tree and checks all fields in each node in order to determine if the node has changed. Just as with the inconsistency detection, the algorithm uses the node name to match two nodes from the trees to each other, and reports if a node has been added, deleted or changed. The difference is that all nodes are traversed regardless of whether the children of a node has changed, as there is no way to determine this without checking the children, and that all fields of a node record are compared one by one individually instead of a fixed-length hash value being compared for the nodes.

In addition, a counter will be implemented that counts the number of nodes traversed for the hash tree, so this number can be compared to the number of nodes that are traversed for the complete traversal approach, which is all nodes in the schema. This will show how much traversal in terms of number of nodes can be saved with the hash tree algorithm.
Chapter 5

Results and Analysis

This section presents results of various performance tests done with the hash tree algorithm compared to the complete traversal approach currently used at Cisco, and an analysis of these results. The first section presents results done on performance tests for specially constructed cases in order to determine the best and worst case performance of the hash tree algorithm. The second section presents performance results on real-life update scenarios that have occurred in the system. This will determine the typical case performance of the hash tree algorithm.

All tests were done on six schemas of varying number of nodes, in order to determine how the number of nodes affects the result and to see if the results can be replicated for different schemas. The three biggest schemas available for testing were used, as it is the bigger schemas that perform the worst for the complete traversal approach. Three smaller schemas were also used to provide some contrast scenarios. The schemas are labeled A to F to distinguish between them. Below the number of nodes for each schema’s latest version of the versions tested is shown.

- Schema A: 443,937 nodes
- Schema B: 190,830 nodes
- Schema C: 115,434 nodes
- Schema D: 9904 nodes
- Schema E: 8590 nodes
- Schema F: 6258 nodes

5.1. Specially Constructed Cases

Since the hash tree algorithm only traverses nodes that have some change in the node hash, the performance depends on how many changes have been made to the schema between versions. In the worst case, the whole schema must be traversed. In the best case, no change has occurred, which means that only the node hash of the root needs to be checked before
the algorithm is done. Since a branch is traversed only until there are no more changes to the branch, the placement of the changes in the tree also affects performance. From these realizations, the following tests have been constructed to determine the effect of each scenario:

- No change to the tree
- One change to a leaf node near the root of the tree
- One change to a leaf node in a long branch
- Changes to all nodes in the tree

5.1.1. No Changes to the Schema

In this scenario, no changes that are relevant to detect for Cisco’s purposes have been made to the schema. In the case of the hash tree algorithm, this will be detected at the root of the hash tree and no further actions have to be taken. This is therefore the best case scenario for the hash tree algorithm, and the algorithm finishes in less than a microsecond. As can be seen in figure 5 below, the time for the hash tree algorithm is set to zero.

For the complete traversal algorithm, the performance is entirely dependent on the number of nodes in the schema and not the number of changes. As the algorithm cannot determine if any changes has occurred further down the tree, the whole tree is traversed no matter the number or positioning of changes. Therefore the number of nodes is the only factor that determines the performance.

Because the performance of the hash algorithm is constant and the performance of the complete traversal algorithm depends on the number of nodes in the schema, the factor of improvement will depend on the size of the schema.

![Figure 5: Performance of hash tree algorithm compared to complete traversal algorithm when no changes has occurred between versions](image)

5.1.2. Change to Shallow Leaf

In this scenario, one change has been made to a leaf node in a shallow branch, in other words, a leaf node near the root of the tree. For the case of the hash tree algorithm, this means that
all nodes down to the change must be traversed, but that this can be expected to be relatively few nodes as the change is made near the root. Therefore, the traversal can be expected to take longer than for the case where no changes have occurred, as some traversal must take place, but it can still be expected to finish in a relatively short time. From figure 6 below, it can indeed be determined that the hash tree algorithm performs significantly better than the complete traversal algorithm in this case. It only takes between 4 and 20 microseconds for the tested schemas, compared to the complete traversal algorithm where it takes 2 milliseconds for the smallest schema.

In the case of the complete traversal algorithm, the whole tree will still be traversed, as the algorithm cannot determine if changes has occurred further down the tree. Figure 6 shows that the result of the complete traversal is the same as the above scenario, as it is still only the number of nodes that determine the performance of the system.

5.1.3. Change to Deep Leaf

In this scenario, a change has been made to the deepest leaf node in the tree. For the hash tree, this means that the longest branch of the tree has to be traversed to the end. This shows the effect the depth of the tree has on a node change. More nodes have to be traversed compared to when a change has been made to a more shallow leaf, as a longer branch contains more nodes to traverse. This case depends on the depth of the tree as well as the number of siblings that need to be traversed on each level. This is because for each node where a child hash differ, all the node’s children must be traversed to detect which child of the children differ, so that it can be determined what sub-tree should be traversed. This however is still expected to only require traversal of a relatively small part of the tree, so the hash tree algorithm should still perform significantly better than the complete traversal approach, which is also shown in figure 7. The results also show that the performance has increased compared to when only one change to a shallow leaf has been made, as it now takes between 11 and 35 microseconds to traverse the tree for the hash tree algorithm, which can be explained by the fact that more nodes must be traversed for a deeper branch.
As before, the complete traversal approach still traverses all nodes in the tree, which means that only the number of the nodes in the schema determine the performance, which also means that, as can be shown in figure 7, the complete traversal approach performs the same as for the previous experiments.

![Figure 7: Performance of hash tree algorithm compared to complete traversal algorithm when one change to a leaf node in a deep branch has occurred between versions](image)

**5.1.4. All Nodes Have Changed**

In this case, both algorithms have been altered to detect changes for all nodes, which means that both algorithms will traverse all nodes in the tree. For the case of the complete traversal approach, this is no different than in previous experiments, and figure 8 shows that the performance is the same. For the case of the hash tree algorithm, this provides a worst case scenario, as usually not all nodes in the tree has changed and some branches therefore do not require traversal. The result is shown in figure 8.

As can be seen in the figure, the hash tree algorithm still performs better than the complete traversal approach used by Cisco. This can be explained by the fact that only three 128-bit hashes are compared for each node instead of several fields of data. Therefore, in the complete traversal approach, more work is done for each node in the tree, which explains why this algorithm still performs worse. However, it is important to note here that because the hash tree algorithm does not check the individual fields of data in the node, in contrast to the complete traversal approach, the hash tree algorithm cannot report what data in the node needs to be updated. In order to determine the individual field changes, the hash tree algorithm would need to compare data fields in the same way as the complete traversal algorithm. If this were to be done in this case where all nodes in the tree have changed, the performance can be expected to be slightly worse than for the complete traversal approach. The reason for this is that in this case, not only do all data fields need to be compared, but also the three 128-bit hash fields for each node. This means that the time it takes to compare the hashes are added to the time it takes to compare the data fields, which would increase the overall time of traversing a schema. However, a scenario where all nodes in the tree have changed is very unlikely.
Other scenarios which would require all nodes to be traversed is if all leaf nodes change or if at least one leaf in a list of sibling leaves has changed for all nodes that are parents to leaf nodes. All branches would then have to be traversed to the end even in the case of the hash tree. In the latter case, this is because all children of a node need to be traversed to determine which of them differ. The children hash can only show that one or more children have changed. In both these cases the hash tree algorithm can still be expected to perform better than the complete traversal algorithm even though the whole tree is traversed, as the data fields only need to be compared when an inconsistency has been detected in the data hash of a node, which is only at the changed leaf nodes. For all other nodes, it is sufficient to check the node’s three hashes to determine that no data has been changed at the node, but that the node’s subtree must be traversed. Therefore, the performance difference shown in figure 8 would be a relatively accurate scenario in this case.

How a data field check affects performance depends on how the hash tree algorithm will be integrated into the current system at Cisco in the finished product. If the updates were to be made to the schema as soon as an inconsistency is detected, as it is done today, the data field check will have the affect described above. As described in section 4.3.1, the hash tree prototype developed for this work returns a data structure showing the differences in the tree and the path to them. If this were to be used in the final product, the branches of the YANG tree corresponding to the returned data structure path would have to be traversed to find and update all differences. This means an extra traversal of the changed branches and data field checks to the changed nodes when performing the updates. This will have an effect on the performance of the complete update procedure, but not on the inconsistency detection performance, which is what has been investigated for this thesis.

![Figure 8: Performance of hash tree algorithm compared to complete traversal algorithm when all nodes have changed between versions](image)

**5.2. Real-Life Update Scenarios**

For these tests, real-life updates that have been done to schemas were used for each schema. For each of the schemas, ten different randomly selected version updates were used. This will show how the hash tree algorithm performs compared to the complete traversal algorithm in typical cases. This gives a more realistic view of the performance difference compared to the
specially constructed cases, as this shows how the algorithm can be expected to perform in real-life scenarios. Tables 1 to 6 below show how many nodes exist in the tree, which is the number of nodes that are traversed for the complete traversal approach, and how many nodes have to be traversed in the hash tree for each schema version chosen. The two versions are two consecutive commits done by the customer. The versions were chosen randomly from the version history for each schema to give an accurate representation of update scenarios. Table 1 to 6 shows the results for schema A to F respectively. The rows in the tables are colored in pairs to show which versions were compared to each other.

Table 1: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema A

<table>
<thead>
<tr>
<th>Version</th>
<th>#nodes total</th>
<th>#nodes traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>322 747</td>
<td>783</td>
</tr>
<tr>
<td>2</td>
<td>322 815</td>
<td>783</td>
</tr>
<tr>
<td>3</td>
<td>293 629</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>293 629</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>422 145</td>
<td>1687</td>
</tr>
<tr>
<td>6</td>
<td>422 255</td>
<td>1687</td>
</tr>
<tr>
<td>7</td>
<td>422 489</td>
<td>2957</td>
</tr>
<tr>
<td>8</td>
<td>423 917</td>
<td>2957</td>
</tr>
<tr>
<td>9</td>
<td>427 317</td>
<td>521</td>
</tr>
<tr>
<td>10</td>
<td>427 383</td>
<td>521</td>
</tr>
<tr>
<td>11</td>
<td>429 329</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>429 329</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>429 449</td>
<td>689</td>
</tr>
<tr>
<td>14</td>
<td>429 621</td>
<td>689</td>
</tr>
<tr>
<td>15</td>
<td>430 195</td>
<td>125</td>
</tr>
<tr>
<td>16</td>
<td>430 745</td>
<td>125</td>
</tr>
<tr>
<td>17</td>
<td>433 685</td>
<td>1713</td>
</tr>
<tr>
<td>18</td>
<td>433 837</td>
<td>1713</td>
</tr>
<tr>
<td>19</td>
<td>443 821</td>
<td>2649</td>
</tr>
<tr>
<td>20</td>
<td>443 937</td>
<td>2649</td>
</tr>
</tbody>
</table>
## RESULTS AND ANALYSIS

Table 2: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema B

<table>
<thead>
<tr>
<th>Version</th>
<th>#nodes total</th>
<th>#nodes traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>151 097</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>151 097</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>151 859</td>
<td>6635</td>
</tr>
<tr>
<td>4</td>
<td>156 009</td>
<td>6635</td>
</tr>
<tr>
<td>5</td>
<td>156 019</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>156 019</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>156 552</td>
<td>6321</td>
</tr>
<tr>
<td>8</td>
<td>156 808</td>
<td>6321</td>
</tr>
<tr>
<td>9</td>
<td>162 860</td>
<td>8128</td>
</tr>
<tr>
<td>10</td>
<td>162 863</td>
<td>8128</td>
</tr>
<tr>
<td>11</td>
<td>164 461</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>164 461</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>165 150</td>
<td>5878</td>
</tr>
<tr>
<td>14</td>
<td>175 856</td>
<td>5878</td>
</tr>
<tr>
<td>15</td>
<td>182 271</td>
<td>287</td>
</tr>
<tr>
<td>16</td>
<td>182 279</td>
<td>287</td>
</tr>
<tr>
<td>17</td>
<td>187 093</td>
<td>234</td>
</tr>
<tr>
<td>18</td>
<td>187 098</td>
<td>234</td>
</tr>
<tr>
<td>19</td>
<td>190 826</td>
<td>306</td>
</tr>
<tr>
<td>20</td>
<td>190 830</td>
<td>306</td>
</tr>
</tbody>
</table>

Table 3: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema C

<table>
<thead>
<tr>
<th>Version</th>
<th>#nodes total</th>
<th>#nodes traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>79 469</td>
<td>290</td>
</tr>
<tr>
<td>2</td>
<td>79 489</td>
<td>290</td>
</tr>
<tr>
<td>3</td>
<td>92 736</td>
<td>11 008</td>
</tr>
<tr>
<td>4</td>
<td>94 773</td>
<td>11 008</td>
</tr>
<tr>
<td>5</td>
<td>103 240</td>
<td>4620</td>
</tr>
<tr>
<td>6</td>
<td>103 030</td>
<td>4620</td>
</tr>
<tr>
<td>7</td>
<td>104 717</td>
<td>35 522</td>
</tr>
<tr>
<td>8</td>
<td>79 489</td>
<td>35 522</td>
</tr>
<tr>
<td>9</td>
<td>106 274</td>
<td>280</td>
</tr>
<tr>
<td>10</td>
<td>106 290</td>
<td>280</td>
</tr>
<tr>
<td>11</td>
<td>108 637</td>
<td>7621</td>
</tr>
<tr>
<td>12</td>
<td>109 087</td>
<td>7621</td>
</tr>
<tr>
<td>13</td>
<td>113 591</td>
<td>213</td>
</tr>
<tr>
<td>14</td>
<td>113 600</td>
<td>213</td>
</tr>
<tr>
<td>15</td>
<td>114 306</td>
<td>2173</td>
</tr>
<tr>
<td>16</td>
<td>114 374</td>
<td>2173</td>
</tr>
<tr>
<td>17</td>
<td>115 332</td>
<td>214</td>
</tr>
<tr>
<td>18</td>
<td>115 362</td>
<td>214</td>
</tr>
<tr>
<td>19</td>
<td>115 430</td>
<td>214</td>
</tr>
<tr>
<td>20</td>
<td>115 434</td>
<td>214</td>
</tr>
</tbody>
</table>
Table 4: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema D

<table>
<thead>
<tr>
<th>Version</th>
<th>#nodes total</th>
<th>#nodes traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3090</td>
<td>448</td>
</tr>
<tr>
<td>2</td>
<td>3100</td>
<td>448</td>
</tr>
<tr>
<td>3</td>
<td>5551</td>
<td>531</td>
</tr>
<tr>
<td>4</td>
<td>5551</td>
<td>531</td>
</tr>
<tr>
<td>5</td>
<td>5639</td>
<td>98</td>
</tr>
<tr>
<td>6</td>
<td>5647</td>
<td>98</td>
</tr>
<tr>
<td>7</td>
<td>8401</td>
<td>263</td>
</tr>
<tr>
<td>8</td>
<td>8431</td>
<td>263</td>
</tr>
<tr>
<td>9</td>
<td>8552</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>8557</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>8825</td>
<td>158</td>
</tr>
<tr>
<td>12</td>
<td>8833</td>
<td>158</td>
</tr>
<tr>
<td>13</td>
<td>9290</td>
<td>125</td>
</tr>
<tr>
<td>14</td>
<td>9302</td>
<td>125</td>
</tr>
<tr>
<td>15</td>
<td>9302</td>
<td>535</td>
</tr>
<tr>
<td>16</td>
<td>9366</td>
<td>535</td>
</tr>
<tr>
<td>17</td>
<td>9603</td>
<td>181</td>
</tr>
<tr>
<td>18</td>
<td>9639</td>
<td>181</td>
</tr>
<tr>
<td>19</td>
<td>9904</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>9904</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema E

<table>
<thead>
<tr>
<th>Version</th>
<th>#nodes total</th>
<th>#nodes traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6351</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6351</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6664</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>6689</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>7098</td>
<td>399</td>
</tr>
<tr>
<td>6</td>
<td>7162</td>
<td>399</td>
</tr>
<tr>
<td>7</td>
<td>7408</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>7408</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>6453</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>6453</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>7551</td>
<td>190</td>
</tr>
<tr>
<td>12</td>
<td>7564</td>
<td>190</td>
</tr>
<tr>
<td>13</td>
<td>7609</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>7609</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>8246</td>
<td>65</td>
</tr>
<tr>
<td>16</td>
<td>8255</td>
<td>65</td>
</tr>
<tr>
<td>17</td>
<td>8309</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>8309</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>8590</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>8590</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 6: Number of total nodes and nodes traversed for the hash tree for 20 versions of schema F

<table>
<thead>
<tr>
<th>Version</th>
<th>#nodes total</th>
<th>#nodes traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5521</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5521</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5523</td>
<td>216</td>
</tr>
<tr>
<td>4</td>
<td>5537</td>
<td>216</td>
</tr>
<tr>
<td>5</td>
<td>6120</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6120</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>5906</td>
<td>199</td>
</tr>
<tr>
<td>8</td>
<td>5912</td>
<td>199</td>
</tr>
<tr>
<td>9</td>
<td>6058</td>
<td>275</td>
</tr>
<tr>
<td>10</td>
<td>6074</td>
<td>275</td>
</tr>
<tr>
<td>11</td>
<td>6092</td>
<td>209</td>
</tr>
<tr>
<td>12</td>
<td>6100</td>
<td>209</td>
</tr>
<tr>
<td>13</td>
<td>6187</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>6187</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>6215</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>6215</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>6220</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>6232</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>6232</td>
<td>228</td>
</tr>
<tr>
<td>20</td>
<td>6258</td>
<td>228</td>
</tr>
</tbody>
</table>

When only one node is traversed for the hash tree there are no changes to the tree, as only the root is checked before finishing.

Figures 9 to 14 show the result of the performance testing on schema A to F respectively for the versions shown in the tables above. Each bar shows the time in microseconds for the inconsistency detection of the complete traversal approach and the hash tree approach between two versions of one schema.

![Figure 9: Real-life update scenarios for schema A](image)
Figure 10: Real-life update scenarios for schema B

Figure 11: Real-life update scenarios for schema C
CHAPTER 5: RESULTS AND ANALYSIS

Figure 12: Real-life update scenarios for schema D

Figure 13: Real-life update scenarios for schema E
The variations in time for the complete traversal algorithm can be explained by the varying number of nodes in the schema between updates. It can be seen from table 1 to 6 together with figure 9 to 14 respectively that the number of nodes in the schema is directly correlated to the time it takes for the complete traversal algorithm to traverse the schema. Later versions often have more nodes added which is why the performance increases in later versions for most schemas.

Figure 15 below shows the relationship between the number of nodes that have to be traversed for the hash algorithm and the time it takes to traverse them. The data for this is gathered data for the number of nodes traversed and the time this takes from all schema runs for the hash tree algorithm.
From this diagram, it can be concluded that there is a linear relationship between the number of nodes that have to be traversed and the performance of the hash tree algorithm. The number of nodes that have to be traversed depends on the number and positioning of the changes to the schemas. Therefore, it can also be concluded that the number and positioning of the changes affect the performance. From the diagram, it can also be determined that most schemas have a very low number of nodes having to be traversed compared to the total number of nodes in the schema. Of the 60 version updates tested, only 13 take longer than 200 microseconds to traverse. When excluding the last value that strays from the others, this becomes more clear, as seen in figure 16.

![Figure 16: Relationship between time and number of nodes traversed by the hash tree algorithm, last value excluded](image)

Table 7 below shows the average number of nodes that have to be traversed for each schema when considering all version updates for the complete traversal algorithm and hash tree algorithm, as well as by what factor the number of nodes that needs to be traversed has been reduced by the hash tree algorithm.

<table>
<thead>
<tr>
<th>Schema</th>
<th>Complete Traversal</th>
<th>Hash Tree</th>
<th>Factor of Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>405 614</td>
<td>1112,6</td>
<td>365</td>
</tr>
<tr>
<td>B</td>
<td>167 575</td>
<td>2779,2</td>
<td>60</td>
</tr>
<tr>
<td>C</td>
<td>104 233</td>
<td>6215,5</td>
<td>17</td>
</tr>
<tr>
<td>D</td>
<td>7824,35</td>
<td>244</td>
<td>32</td>
</tr>
<tr>
<td>E</td>
<td>7433,45</td>
<td>79</td>
<td>94</td>
</tr>
<tr>
<td>F</td>
<td>6011,5</td>
<td>113,2</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 8 below shows the average time to traverse each schema when considering all version updates for the complete traversal algorithm and hash tree algorithm, as well as by what factor the traversal time has been reduced by the hash tree algorithm.
Table 8: Average traversal time for complete traversal and hash tree approach as well as the factor of improvement in time for each schema

<table>
<thead>
<tr>
<th>Schema</th>
<th>Complete Traversal</th>
<th>Hash Tree</th>
<th>Factor of Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>125 085.85</td>
<td>194.1</td>
<td>644</td>
</tr>
<tr>
<td>B</td>
<td>50 127.25</td>
<td>507.1</td>
<td>99</td>
</tr>
<tr>
<td>C</td>
<td>29 654.05</td>
<td>1031.0</td>
<td>29</td>
</tr>
<tr>
<td>D</td>
<td>3699.9</td>
<td>37.05</td>
<td>100</td>
</tr>
<tr>
<td>E</td>
<td>3270.45</td>
<td>7.7</td>
<td>424</td>
</tr>
<tr>
<td>F</td>
<td>2490.15</td>
<td>18.25</td>
<td>136</td>
</tr>
</tbody>
</table>

Both these tables show that for all schemas used in this experiment, a significant factor of improvement occurs when using the hash tree algorithm. Table 7 shows that the hash tree algorithm only needs to traverse on average 1.5% of the number of nodes the complete traversal algorithm does. For the worst case schema, schema C, there is still a substantial improvement, with only 6% of the total number of nodes having to be traversed for the hash tree. This means that even in the worst cases the hash tree algorithm can be expected to perform significantly better than the complete traversal algorithm.

When comparing the average improvement in time shown in table 8, the hash tree algorithm can be expected to yield an average of a 238 times improvement, with approximately an average of a 30 times improvement for the worst case schema, schema C. The worst case version update among the schema updates tested, which is update 7 to 8 in schema C, showed a five times performance improvement for the hash tree. This is still a significant improvement.

From the tables above, it can be calculated that the complete traversal algorithm can traverse on average approximately 3.3 nodes per microsecond, while the hash tree algorithm can traverse on average 5.9 nodes per microsecond. This means that it takes approximately half the time for the hash tree algorithm to check a node compared to the complete traversal algorithm. The reason for this, as discussed in section 5.1.4, is because the hash tree only compares hashes of node data instead of the actual node data. This is the third factor together with the number and positioning of changed nodes that determine the improvement in performance. However, as mentioned in 5.1.4, it is important to note that the hash tree algorithm does not distinguish which node fields differ for a differing node. If the hash algorithm were to be changed when integrated with the current system at Cisco so that it would detect which fields of data must be updated, this would mean that for the nodes that have changed, the algorithm must compare the three hashes of the node as well as the node’s data fields. This is however only true for nodes that have a change to their data hash, as only these nodes have changed themselves.

When looking at the number of nodes traversed for the hash algorithm, 16 of the 60 version comparisons showed no difference to the schemas. This means that in approximately 27% of cases, only the node hash of the root has to be checked, which is a 128 bit hash. It can therefore be assumed that a significant portion of version updates will contain no differences, and therefore not only will save time because the schema does not have to be traversed, but also as no update procedure is required to update the database schema version in CDB.

5.3. Discussion

The original question this thesis aimed to answer was whether a hash tree algorithm could improve the performance of the database schema inconsistency detection compared to an algorithm that traverses all nodes in the schema tree. From the results, it is clear that the
hash tree algorithm improves the performance considerably in the best and typical cases. It can therefore be concluded that an update to using a hash tree algorithm instead of using a complete traversal approach would be beneficial.

### 5.3.1 Sources of error

A possible factor that might affect the result of the inconsistency detection performance measurements for both the hash tree and complete traversal algorithms is how the caching is handled by the operating system. When first running the tests, for some schemas the results were very varied, varying from a few tens of microseconds to tens of thousands of microseconds. After doing extensive troubleshooting to find out what could cause this uneven result, it was discovered that the reason was caching. This was discovered by testing a method called cache warming. This entails loading the dataset into the cache, in this case by running the inconsistency detection once before running it while measuring the time. This will ensure that when the program is run while the time is measured, cache hits will occur on the data. This gives an even result as the system gets constant hits in the cache. However, this means that the times measured and presented in the result relies on the data to be in the cache to be accurate. How the time is affected with cache misses is therefore not measured.
Chapter 6

Conclusions and Future Work

This section presents a summary of what has been done and the results of the work, as well as conclusions that can be drawn. The section also includes a discussion on the limitations of the work and results, and a section on future work.

6.1. Conclusions

For this thesis, two algorithms were developed that can be used together to detect inconsistencies in data. The first algorithm takes an existing tree data structure, in the case of this work, a YANG schema, and converts it into a hash tree. Each node in the hash tree consist of three hashes to determine if some data change has happened to the current node and if a change has happened in the node’s subtree. To see if a change has occurred in the subtree, one hash is a combination of the hashes of the node’s children nodes. In ordinary Merkle trees, the combined hash of a node’s children is created by joining the hashes of the children, for example by hashing the child hashes in order. This means that if the positioning of the children nodes changes, the combined hash will change. For this work, changes to children’s positioning should not constitute as a change to the data, and therefore the combined hash should not change in this case. Therefore, instead of joining the hashes, bitwise XOR was used to XOR the hashes together. As XOR is commutative, this will not affect the final hash.

The results show that performance can be expected to be drastically improved by using hash trees for database schema inconsistency detection compared to using an approach that traverses all data in the schemas regardless of need. The results also show that for the hash tree algorithm, the level of improvement depends on the number of differences as well as the positioning of the differences in the tree. This is because these factors determine how many nodes need to be traversed in the hash tree. In this case, the total number of nodes in the tree does not have a direct effect on performance, as the level of traversal only depends on the number and positioning of nodes that have changed, independent of the total number of nodes. It does however have an effect in the cases where all branches of the tree need to be traversed, as a bigger schema means more to traverse in this case.

For the complete traversal approach, the total number of nodes in the schema determines the performance, whereas the number or positioning of the changed nodes have no effect. This can be explained by the fact that the whole schema is traversed regardless of the changes made to the schema, as there is no way to determine the need to traverse the children without actually traversing the children.
In the typical case, few changes have occurred between schema versions, with on average only 1.5% of the total number of nodes needing to be traversed for the hash tree algorithm. The performance tests done on real-life update scenarios show that in 78% of cases the hash tree algorithm finishes in under 200 microseconds. This is illustrated in figures 15 and 16 that show the relationship between the number of nodes that need to be traversed and the time it takes to traverse them. For the worst case schema, the schema with the worst performance, the hash tree algorithm performed on average 30 times better than the complete traversal algorithm. The worst case version update among the schema updates tested showed a five times performance improvement for the hash tree, which is still a significant improvement.

In the best case, no changes has occurred and the time to detect inconsistencies is instantaneous, as the program only has to compare one 128 bit hash before finishing. The results show that this can be expected to happen in 27% of the cases, which is more than a fourth of all cases. The performance is further improved in this case by the fact that no updates have to be made to the schema, so the procedure that updates the schema in CDB does not need to run.

In cases with one change to a leaf near the root, the algorithm only needs a few microseconds to detect the inconsistency. Cases where one change has occurred to a leaf in a deep branch takes longer than when the change has occurred to a leaf near the root, but the time it takes is still insignificant. In both these cases, the time depends on the depth of the branch the leaf is part of, and the number of siblings to traverse for each level until the leaf is reached. When a change has occurred to a leaf in a deeper branch, more nodes must be traversed to reach the inconsistency, which is why this takes longer than a change to a leaf in a shallow branch.

In the worst case, the whole tree must be traversed even in the case of the hash tree algorithm. This happens if all nodes have changed, all leaf nodes have changed or at least one leaf in a list of sibling leaves has changed for all nodes that are parents to leaf nodes. This would be a very unlikely occurrence. In this case, the algorithm has an improvement factor of about 2. It is important to remember here however that the hash tree algorithm does not detect which individual fields in the node has been changed. If it had, the performance can be expected to be slightly worse than the complete traversal approach in the case of all nodes changing, but still about twice as fast in the other two cases that require all nodes to be traversed, as the data fields only need to be checked if the data hash differs for a node.

### 6.2. Limitations

When measuring performance, the solution was compared to a system written that performs similar actions to the system used by Cisco today. The reason for this, in contrast to measuring the performance difference on the actual system, is because the integration of the solution to the existing system is out of the scope for this project partly due to time constraints and partly due to the fact that measuring the performance of the inconsistency detection alone in the current system would be challenging, as the inconsistency detection and updating the discovered inconsistencies is integrated in the current system. That is why a separate system mimicking the inconsistency detection of the current complete traversal approach used by Cisco was built. This allows for measuring the relative performance difference in order to get an understanding of how much hash trees can be expected to improve the performance compared to the current method of complete traversal. However, this is only an approximation of the performance improvement. The numbers are therefore not an exact representation of how a hash tree algorithm would perform when integrated in the current system. Nevertheless, the numbers do show a general performance improvement.
between inconsistency detection with and without hash trees, which in itself is enough to answer the question formulation.

### 6.3. Future Work

The work that was done for this thesis is limited by comparing a prototype of a hash tree inconsistency detection algorithm and an algorithm intended to simulate the current method of inconsistency detection used at Cisco. This limitation means that the results are estimations of how the performance will be affected by a change to hash trees. In order to measure the true effect on performance, as well as actually making the system useful for the product, the algorithms developed for this project will need to be integrated to the current overall system at Cisco. When this is done, further performance tests need to be performed to determine the actual improvement of performance.

When integrating the system to Cisco it might be beneficial to perform the update procedure as soon as an inconsistency is detected. As it is now, the algorithm reports the result in a tree of differing nodes and the path to them (see section 4.3.1). The original YANG tree will then have to be traversed according to this path. If instead the update was to be performed immediately, the path to the differing nodes would only need to be traversed once; when the inconsistencies are first detected.

Yet another aspect to consider is that only one structure for the hash tree was investigated, which was to preserve the YANG tree structure of the hash tree. An alternative would be to instead serialize the YANG tree to a linear array of blocks and build a standard binary Merkle tree from those data blocks. This would require some system to keep track of the tree structure for when the updates are to be made to the original YANG schema, which is why this approach was not chosen. However, possible future work could be to investigate this possibility and measure the performance of a tree structured in this way to the structure chosen for this project. If it was to be shown that the serializing the data is more effective, it might be worth considering this approach instead. Furthermore, there are other possible structures to use for the hash tree that has not been considered and that could be investigated as future work.
References


