MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Updates of distributed provenance chains with integrity verification

Bachelor's Thesis

RADOMÍR MANN

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Updates of distributed provenance chains with integrity verification

Bachelor's Thesis

RADOMÍR MANN

Advisor: Mgr. Rudolf Wittner

Department of Computer Systems and Communications

Brno, Spring 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

I declare that I have worked on this thesis independently, using only the primary and secondary sources listed in the bibliography

Radomír Mann

Advisor: Mgr. Rudolf Wittner

Acknowledgements

I want to thank and express gratitude to my advisor Mgr. Rudolf Wittner, for the time and effort devoted to guidance and assistance while I have been writing this bachelor thesis.

Abstract

Provenance is information that describes an object's origin and can be used to evaluate its properties like quality, reliability, and trustworthiness. Provenance about a single object may be distributed across independent locations. Interconnected provenance stored in different areas forms distributed provenance chain. These chains enable the discovery and acquisition of an object's provenance spread in several places. This thesis aims to determine how updates of individual provenance parts and their integrity verification may affect an object's provenance traceability represented in distributed provenance chains and demonstrate cases during the search for an object's provenance. When a case reveals some obstacle, I propose a solution that resolves this problem. At last, I present a prototype that can search an object's provenance in a distributed environment with updates and integrity verification while solving different problems demonstrated by various cases.

Keywords

integrity, provenance, security, distributed provenance, Python

Contents

Introduction 1

1 Introduction to provenance data model 3
- 1.1 W3C PROV and data model . . . 3
- 1.2 Provenance structures . . . 3
  - 1.2.1 Entity . . . 4
  - 1.2.2 Activity . . . 4
  - 1.2.3 Agents . . . 5
- 1.3 Identification of structures in W3C PROV . . . 5
- 1.4 Extensibility points . . . 6
- 1.5 Provenance document . . . 6
- 1.6 Provenance bundle . . . 6

2 Distributed provenance 7
- 2.1 Distributed provenance structure . . . 7
- 2.2 Bundle versioning . . . 8
- 2.3 Extensions of versioning . . . 9
  - 2.3.1 Merge of bundles . . . 9
  - 2.3.2 Fork of bundles . . . 10

3 Provenance integrity and search algorithm 12
- 3.1 Integrity in provenance data model . . . 12
  - 3.1.1 Provenance integrity . . . 12
  - 3.1.2 Key management . . . 13
- 3.2 Search for entity's relevant provenance in distributed provenance chains . . . 13
  - 3.2.1 Simulation of distributed environment . . . 13
  - 3.2.2 General search algorithm . . . 14
- 3.3 Influence of integrity verification on the search algorithm 17
  - 3.3.1 Integrity of provenance chains . . . 17
  - 3.3.2 Integrity of meta-bundle . . . 18
  - 3.3.3 Search modes . . . 19

4 Possible cases of extended search algorithm 20
- 4.1 Semantics and simplification of graphs . . . 20

4.1.1 Simplification of graphs 20
4.1.2 Colours of bundles 22
4.2 Cases without updates 23
4.2.1 Basic cases 23
4.2.2 Cases with backward references 28
4.2.3 Cases with integrity validation 29
4.3 Linear updates 31
4.3.1 Basic linear updates 31
4.3.2 Linear updates with backward references 33
4.3.3 Linear update with integrity verification 35
4.4 Cases with merges and forks 38
4.4.1 Merge cases 38
4.4.2 Fork cases 40
4.5 Updates in cycle 42

5 Implementation **43**
5.1 Overview of the required functionality 43
5.2 Python library 45
5.3 Provenance generator 45
5.3.1 Generator functionality 45
5.3.2 Generator input 47
5.4 Integrity verification 48
5.4.1 Hashing of bundles 48
5.4.2 Content of token 48
5.5 Structure of the implementation 49
5.5.1 Search 49
5.5.2 Generator 49
5.5.3 Utilities 49
5.5.4 LinkedList 49
5.5.5 Crypto 50
5.5.6 Test 50

6 Results and conclusion **51**

Bibliography **53**

List of Figures

1.1 Representation of record in PROV-N notation 3
1.2 Intuitive overview of core structures [5] 4

2.1 Distributed provenance chain 7
2.2 Representation of update in meta-bundle [3] 9
2.3 Expression of the merge operation in meta-bundle 10
2.4 Expression of the fork operation in meta-bundle 11

4.1 Simplification of relations in graph 21
4.2 Meaning of bundle's colours within graphs 22
4.3 Basic case without has provenance 23
4.4 Basic case without interconnections of chains 24
4.5 Basic case with content and no interconnections 25
4.6 Basic case with interconnections 26
4.7 Basic case with interconnections and content 27
4.8 Has provenance cycle 28
4.9 Special case of cycle 28
4.10 Has provenance cycle with content 29
4.11 Integrity without interconnections of chains 29
4.12 Integrity with interconnections of chains 30
4.13 Integrity with interconnections and explicit content 31
4.14 Basic linear update 31
4.15 Basic linear update without entity in newer version 32
4.16 Basic linear update without entity 33
4.17 Backward reference into older version 33
4.18 Backward reference into older version with different entity 34
4.19 Linear update with reference into newer version 34
4.20 Invalid older version in update branch 35
4.21 Invalid between older and newer version 36
4.22 Reference into invalid bundle with update 37
4.23 Multiple references into update branch with invalid bundle 38
4.24 Merge of bundles 39
4.25 Merge with content and invalid bundles 39
4.26 Fork of bundle with invalid bundles 40
4.27 Fork with predecessors with multiple newer bundles . . . 41
4.28 Document with updates in cycle 42

5.1 Generator activity diagram 46

Introduction

The modern digital world provides a simple way to exchange data. However, data shared on the Internet are not necessarily trustworthy, and their quality and reliability can be questionable. People usually try to assess data based on author, location (website, reliable organization), sources, involved processes (processes that participated in data generation) or broader context. The aforementioned information can be referred to as provenance.

"Provenance is information that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering specific objects." [1]. Described objects do not have to be only formed from digital data but can even represent physical and intellectual things [2]. Provenance may be used to assess a particular object's quality, reliability, and trustworthiness. It may hold people responsible in its generation to be able to give them credit when reusing it [1]. Organizations or people creating objects can generate related provenance and attach it to the described object to increase its credibility. Different independent organizations or people can later use this object to produce a new one. The provenance of this new object may contain references to a related provenance about things utilized in the generation. These references make gathering provenance about objects that have been used throughout the creation process feasible. Interconnected provenance generated in different places forms distributed provenance chains.

Modification of the provenance can change its semantics, which is essential during its evaluation. Therefore protection against unauthorized modification of provenance is required to guarantee provenance integrity. Hashes and digital signatures can be used to prove that data are in the original form created by a specific source.

Provenance modification can be authorized when an author wants to repair a mistake or extend an object's provenance. An authorized modification is called an update. Update of records must not disrupt the provenance chain integrity, foreign references, and auditability of changes[3]. As a result versioning mechanism that preserves these properties is essential.

The thesis's primary goal is to analyze how authorized and unauthorized changes can affect the traceability of an object's provenance

in distributed provenance chains and show possible situations during the traversal. On top of that, the thesis suggests how to realize integrity verification of provenance parts within distributed chains. The last goal of this thesis is to implement the prototype in the python PROV library that shows the search algorithm with integrity verification while solving different obstacles demonstrated in the thesis.

1 Introduction to provenance data model

This chapter introduces the data model defined within the current standard for provenance representation.

1.1 W3C PROV and data model

W3C PROV is a general-purpose standard that defines the way to express provenance. W3C PROV consists of multiple electronic documents [2]. PROV-DM document [1] describes the data model. The data model unifies the representation of information captured in provenance to enable sharing between independent systems and organizations [1].

1.2 Provenance structures

Provenance structure is an element of the data model that represents the concrete part of provenance captured about objects [1]. A structure has a name and attributes that describe an object itself, its properties or relations. Attributes are separated into three categories: mandatory, optional and additional. Additional attributes add further meaning to the structure and consist of a name and corresponding value [1].

Provenance records are a partial representation of provenance structures. Therefore one structure can be described by multiple records [1]. The figure 1.1 illustrates an example of a provenance record in the standard specific notation PROV-N [4].

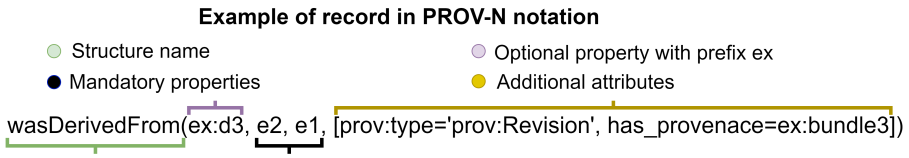

Figure 1.1: Representation of record in PROV-N notation

The following example shows a simple overview of core structures and basic relations between them within the data model.

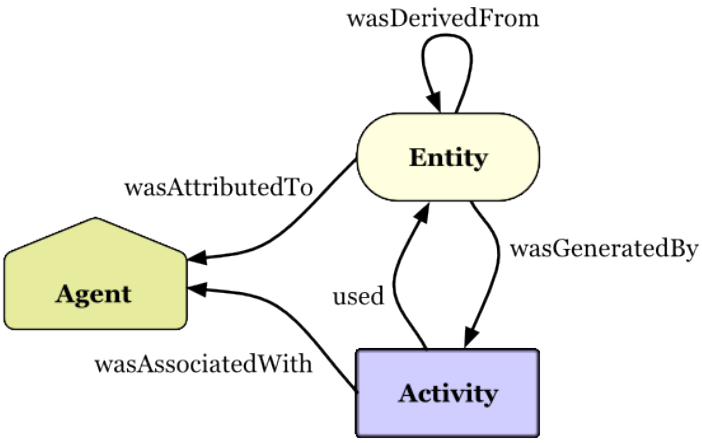

Figure 1.2: Intuitive overview of core structures [5]

1.2.1 Entity

Entity is a provenance structure that represents digital objects (such as files, web pages, digital tokens), physical things (buildings, cars), or intellectual properties (concepts, ideas) [1].

The fundamental relation between entities is derivation. Derivation between two entities represents that entity was somehow influenced by the other entity [1]. For example *"transformation of an entity into another, an update of an existing entity, or the construction of a new entity based on a pre-existing entity"* [1]. Another relation between entities is specialization-of. Specialization-of captures the relationship between a general and specialized entity where the specialized entity shares the same aspects as a general entity but contains more specific or additional information about described object [1].

1.2.2 Activity

Activity is a provenance structure that somehow works with entities over a period of time [1]. More specifically, it is *"consuming, processing, transforming, modifying, relocating, using, or generating entities"* [1].

The data model connects activity with a utilized entity by usage relation [1]. Usage also captures the start of the utilization of a specific entity by activity [1]. WasGeneratedBy is another relation that denotes the completion of an entity generation by specific activity [1]. As

a result, it also captures activities that participated in the creation process of a particular entity.

1.2.3 Agents

Agent is a provenance structure that represents organizations, people or automated programs that were somehow involved in the creation of objects described by provenance and are responsible for some activities that have taken place [1].

Agents responsible for an entity are linked by attribution relation [1]. The relation that assigns an agent participating in a certain activity is association [1]. Agents can be responsible for the delegated work(activity) outcome. Delegation relation captures this responsibility [1].

1.3 Identification of structures in W3C PROV

W3C PROV uses qualified names to identify provenance structures. Qualified names consist of a prefix and a local-part (prefix:local-part). The prefix is a shorter representation of the namespace where a local-part is located. Namespace groups identifiers and helps avoid collisions in separate domains. Namespace declaration creates a connection between prefix and namespace. Local-part identifies an object within the namespace, and therefore different structures cannot have the same local-part across the single namespace [6]. A namespace is an instance of URI (Uniform Resource Identifier), a unique global identifier of a particular resource. As a result, namespaces with valid URI enable lookup for specific provenance on the Internet.

1.4 Extensibility points

The standard defines extensibility points that allow provenance designers to specialize the data model for various domains and applications [1]. Extensibility points are reserved additional attributes: prov:type, prov:role and prov:location. The most important for this thesis is prov:type. Prov:type provides more application-specific typing for any structure in the data model [1]. For example, a person working as a reviewer within a news agency will be represented in PROV-DM as an agent with additional attribute prov:type with value reviewer [1]. W3C PROV also predefines several standard specific types, such as prov:person, prov:bundle, and prov:revision [1].

1.5 Provenance document

Provenance document is any object that stores provenance records. The most natural representation of a provenance document is the serialization of provenance records into a file, usually in XML, JSON, or standard specific notation PROV-N [4]. More advanced serialization can create records suitable for graph databases.

1.6 Provenance bundle

Provenance bundle is a named container that packs provenance records together. Bundles can define their namespace for inner records [1]. The standard forbids two bundles with the identical name within the same provenance document [6]. Additionally, the bundles are prohibited from containing other bundles [6]. Bundles need to be represented as the entity to enable creating provenance about them [1].

2 Distributed provenance

This chapter describes a representation of distributed provenance chains using the data model and how an existing method for provenance versioning is integrated within these chains [3]. Additionally, I will define special updates that the data model can capture but are not addressed in the existing mechanism for provenance parts versioning within distributed chains [3].

2.1 Distributed provenance structure

Object's provenance or its parts may be created and stored separately, resulting in spreading provenance about this object into several independent locations. Related provenance should be traceable [7] to properly evaluate the object's origin and assess its properties from every piece of available provenance.

Entities representing specific objects may save supplementary information within additional attributes. These attributes are suitable to store links that reference resources containing an object's provenance stored in different locations. The attribute with this functionality is called `has_provenance` [8]. Provenance records are kept within bundles for more straightforward sharing and management. Therefore `has_provenance` contains a resource link (URI) and bundle identifier with related provenance.

Objects utilized in generation could have also been derived from other objects in their production. As a result, references may form a transitive provenance chain. The figure 2.1 shows an example of the interconnection between bundles.

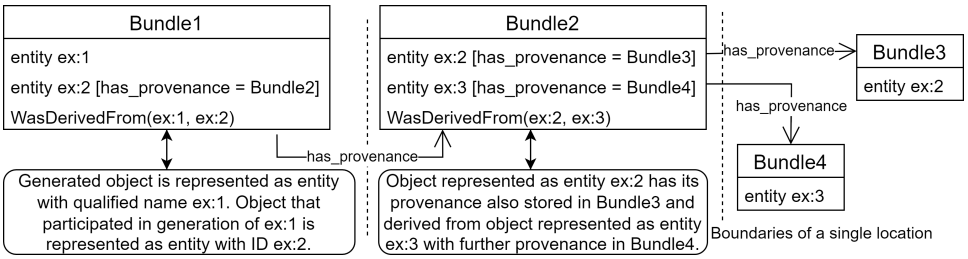

Figure 2.1: Distributed provenance chain

2.2 Bundle versioning

Bundles may contain errors or have missing information. Therefore there is a necessity to be able to update provenance. However, changes cannot be directly applied due to the demand to preserve the provenance chain integrity, foreign references, and auditability of changes. For that reason, there is a need for a suitable versioning mechanism which complies with these requirements.

This mechanism has been already designed [3]. The mechanism does not update a bundle directly but creates a new bundle with modified content while preserving an older version. W3C PROV data model then captures the relations between the newer and older versions of a bundle. These relations are stored within a distinct bundle called meta-bundle. Meta-bundle gathers meta-provenance (provenance about provenance) about bundles [3]. According to the existing mechanism, the meta-bundle must contain entities that represent every bundle within the administrative boundaries of a single subject [3]. Therefore appropriate meta-bundle must capture even bundles with only a single version [3].

Initially, the meta-bundle contains two entities to represent a single version. The first entity represents a base bundle(bundle with independent content on a version), and another entity represents a specific version of that bundle. After an update, a newer version of a bundle containing updated records is created with a corresponding entity in the meta-bundle. The relationship between the older and newer version of a bundle is captured in the meta-bundle by revision. Revision is a relation defined in W3C PROV as extended derivation that uses predefined type prov:revision. The update also connects entities representing the new version of a bundle and its corresponding base bundle by specialization-of relation in the meta-bundle. As a result, every base bundle form one **branch** of linear updates for a particular bundle.

Newer versions of a bundle referenced from another bundle can be found by relations stored in the meta-bundle. As a result, the meta-bundle must be available and accessible to find more recent versions.

The update mechanism creates an updated bundle while preserving the older version traceable in meta-provenance. Therefore, it is still possible to find different versions of a particular bundle in meta-

provenance and see changes that occurred in a specific update [3]. Figure 2.2 depicts the versioning mechanism of one update in a single branch.

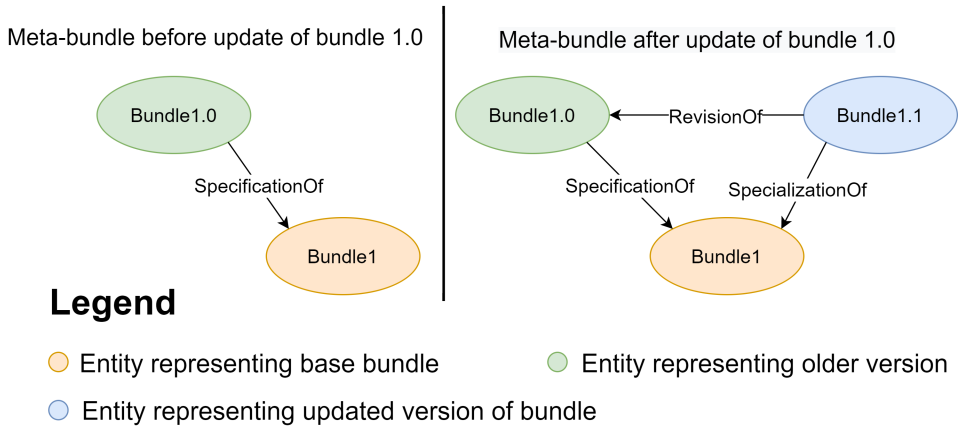

Figure 2.2: Representation of update in meta-bundle [3]

2.3 Extensions of versioning

The existing update mechanism allows only linear versioning of bundles [3]. However, meta-provenance can capture more complex relations between bundles. It can document an operation called merge where content from two or more bundles join into a single bundle or the inverse approach called fork where the content of one bundle split into several others. Merges and forks change the latest version of a bundle and can be seen as an update. Therefore, I will extend the defined update mechanism [3] to support merges and forks.

2.3.1 Merge of bundles

Merge is a type of update where two or more bundles join into one new bundle. Merge can be used when something or someone managing bundles wants to assemble provenance from multiple sources into a single bundle. Merge can be captured in the meta-bundle like a linear update defined earlier with some notable changes.

1. To document the merge, a new base bundle will be created by extracting information from every base bundle from joined

branches. Consequently, the new base bundle independent of the version represents a new update branch for linear updates in the meta-bundle. In other words, all newer linear versions of the bundle produced by the merge will be in the meta-bundle connected by specialization-of relation to entity denoting this recently created base bundle.

2. The entity representing the new bundle(created during merge) is in a meta-bundle connected by a revision to the entities that constitute the latest bundles of merged branches.

The main difference between the merge and linear update is that a newer version has a different base bundle derived from all precedent base bundles. Figure 2.3 shows how the merge changes the meta-bundle.

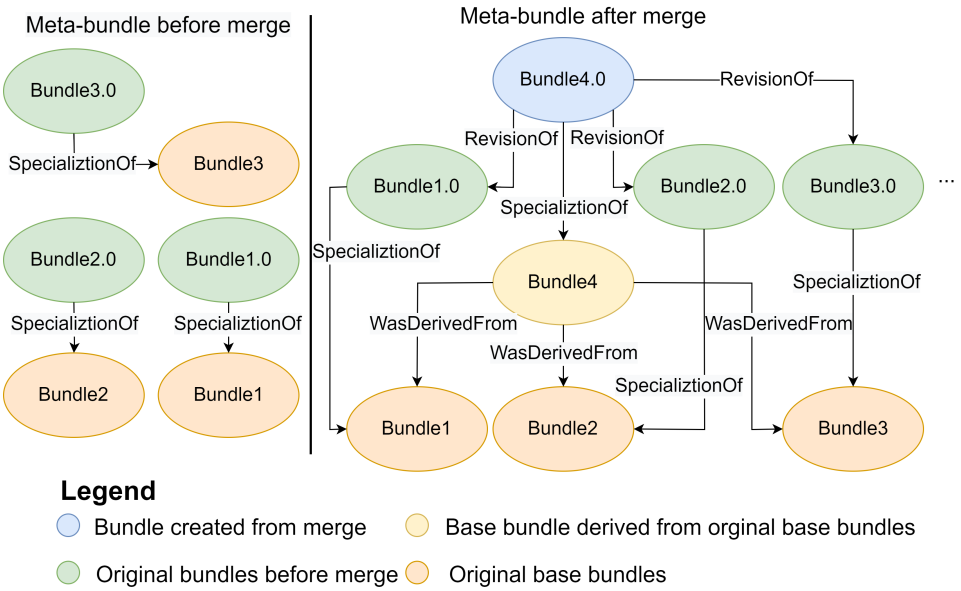

Figure 2.3: Expression of the merge operation in meta-bundle

2.3.2 Fork of bundles

Inverse operation to merge is the fork. A fork splits a single bundle into two or more bundles. Possible use cases are creating multiple update branches from one provenance bundle and reversing merge.

Every bundle originating from a fork has a new base bundle that extracts relevant data from the predecessor's base bundle. Hence the fork creates multiple new update branches from a single update branch. Also, every bundle created from the fork is a revision of the predecessor in a meta-bundle. More illustrative is Figure 2.4, which shows the general fork in the context of the data model in the meta-bundle.

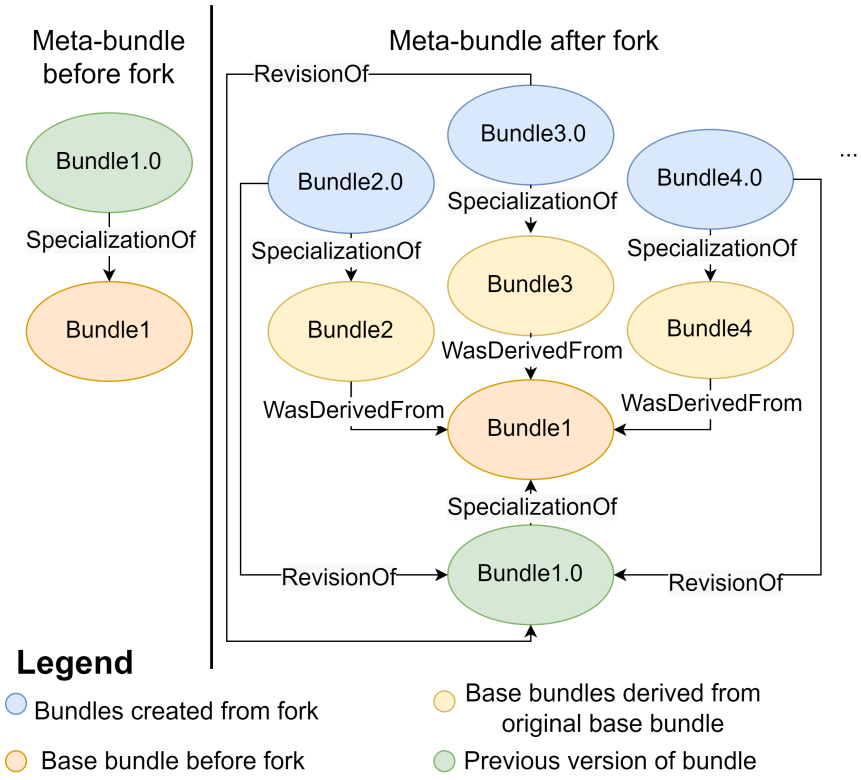

Figure 2.4: Expression of the fork operation in meta-bundle

3 Provenance integrity and search algorithm

This chapter describes a method to verify the integrity of bundles and a general algorithm that searches for relevant provenance of a specific entity within distributed provenance chains. Furthermore, this chapter introduces problems in searching in an environment with integrity verification of provenance parts.

3.1 Integrity in provenance data model

Provenance needs to be protected from unauthorized modification to preserve its trustworthiness. This section shows how I achieved the integrity of provenance using the data model.

3.1.1 Provenance integrity

Data has the property of integrity if there are assurances whether data originates from a particular source along with validation that they are in the unmodified form issued by the creator [9].

Provenance integrity is demanded because it increases the reliability and trustworthiness of captured information [10]. Bundles are provenance parts practical to be protected against unauthorized modification because they are used in distributed provenance and pack provenance records together. I will use combinations of digital signatures and hashes to achieve the integrity of bundles.

When a new version of a bundle is issued, the liable subject creates a token for this bundle. The token contains a digitally signed hash of the bundle and all necessary information to detect possible alternations, such as the public key and resulting signature.

The bundle and its token must be accessible to verify integrity properties. The straightforward approach to associate bundle and its token is by derivation relation within meta-bundle. An entity represents a token in the meta-bundle using extensibility point "prov:type" with value "token". This extensibility point distinguishes a cryptography token derived from a bundle and any other utilized entity.

3.1.2 Key management

Digital signs use public-secret key pairs. The secret key must be protected from its disclosure. With knowledge of the secret key attacker can create a valid signature for any data. In addition, the verification procedure must acquire a public key from a reputable source or be able to verify that a public key has not been changed (for example, by a certificate of public key signed by a trusted certification authority). Otherwise, a potential attacker can switch a public key for another where he/she knows a related secret key and can replicate any signature with different data. However, in this thesis, I will pack a public key within the token in plain form without any verification to simplify its acquisition regardless it is insecure.

3.2 Search for entity's relevant provenance in distributed provenance chains

Provenance about a single object can be spread across multiple bundles located in different places. These bundles are connected by references that form distributed provenance chains. This section mainly describes a general algorithm that can be used to traverse distributed provenance chains to look up bundles containing provenance information related to an entity with a given ID.

3.2.1 Simulation of distributed environment

The thesis demonstrates the search for bundles in distributed provenance chains. Search in distributed chains traverse across multiple locations managed independently. This distribution is not feasible to present in a bachelor thesis. Therefore there is a need to simulate distributed provenance chains. For this reason, I will use abstraction, where a single provenance document with bundles represents the provenance managed by one subject. `Has_provenance` attribute will refer to other bundles stored in other provenance documents instead of bundles possibly located in different areas. This abstraction is suitable because it only affects the means of data acquisition and the structure of the `has_provenance` attribute. These differences do not influence the results of this thesis.

3.2.2 General search algorithm

This section defines the general algorithm that searches for bundles with relevant provenance of a particular object in linear provenance chains (without cycles). More precisely algorithm finds all bundles referenced by `has_provenance` from entities representing a specific object or other objects utilized in its generation. This algorithm provides the starting point of the thesis, and its functionality will be gradually extended to support integrity verification and the versioning mechanism described above.

The algorithm's input is an initial provenance document where the search algorithm will start and the ID of an entity for which the algorithm will discover bundles with related provenance. At first, the algorithm seeks bundles in the initial document containing the searched entity (an entity with an ID passed by a user on the algorithm's input). These bundles are called initial bundles from which the algorithm follows every `has_provenance` reference in the searched entity itself and all entities from which the searched entity was derived.

In the second phase algorithm searches for all bundles referenced by `has_provenance`. Whenever the algorithm processes a bundle discovered by `has_provenance` reference from an entity, it attempts to find an entity with the same ID in the referenced bundle and searches for all entities from which it is derived. Then it follows references from these entities together with ones from the referenced entity (the entity that the algorithm has used to traverse into this bundle). Since these entities represent objects that somehow influenced parts of the object later used to produce the searched object(object represented by the searched entity). This process repeats until the algorithm has found all transitive `has_provenance` references from particular entities within discovered bundles. The algorithm then outputs identifiers of bundles found by the algorithm.

This algorithm simulates the search for an object's provenance within bundles in a provenance chain formed by separate documents. Pseudo-code below 1 shows this algorithm, including auxiliary functions in python-like syntax.

Algorithm 1 General search algorithm pseudo-code

```
procedure BasicSearch(searchedId, startDocument)
    result, bundles ← [], []
    for bundle in startDocument.bundles do
        for entity in bundle do
            if entity.id == searchedId then
                bundles.append((bundle, searchedId))
                break
            end if
        end for
    end for
    while bundles is not empty do
        bundleWithId ← bundles.pop()
        bundle, referencedId ← bundleWithId
        if bundle not in result then
            result.append(bundle)
        end if
        for entity in bundle.entities do
            if entity.id == referencedId or
             IsDerivedFrom(referencedId, entity.id) then
                for attribute in entity.attributes do
                    if attribute.key == "has_provenance" then
                        nextBundle ← GetBundle(attribute.value)
                        if nextBundle is None then continue
                        end if
                        bundles.append((nextBundle, entity.id))
                    end if
                end for
            end if
        end for
    end while
return result
end procedure
```

Algorithm 2 Auxiliary function that test if entity (generatedId) is derived from other entity(usedId)

```
procedure IsDerivedFrom(generatedId, usedId)
    for relation in bundle.relations do
        if relation.type == Derivation and
        relation.generatedEntity == generatedId and
        relation.usedEntity == usedId then
            return True
        end if
    end for
return False
end procedure
```

Algorithm 3 Auxiliary function that extract bundle from different file

```
procedure GetBundle(value)
    file ← open(value.file)
    if not file then return None
    end if
    for bundle in file.bundles do
        if bundle.id == value.id then return bundle
        end if
    end for
return None
end procedure
```

3.3 Influence of integrity verification on the search algorithm

Extension of the general search algorithm by integrity verification may affect its behaviour. This section introduces how integrity verification divides the output into three categories and influences the result of the search algorithm together with the update mechanism.

3.3.1 Integrity of provenance chains

The general algorithm now verifies the integrity of referenced bundle by lookup for a token in a corresponding meta-bundle. When the algorithm traverse reference into a new bundle, it immediately acquires the corresponding meta_bundle, where it seeks token and later also more recent versions. The token is then used to verify the integrity of records within the bundle.

A bundle with an unverifiable or missing token is in output marked as invalid[1]. The algorithm cannot verify that a `has_provenance` reference was not tempered within an invalid bundle. Therefore, bundles referenced after an invalid bundle in a provenance chain are considered untrustworthy because they could have been redirected from the truthful result. Consequently, every bundle referenced by `has_provenance` after an invalid bundle is in the output category with low credibility. As a result, every bundle found by the search algorithm is separated into one of the three distinct categories.

1. Valid bundles are bundles with successfully validated integrity, and every bundle on a path formed by `has_provenance` references from an initial bundle(included) to this bundle is also valid.

2. Invalid bundle has an unverifiable or missing token, and therefore its content cannot be trusted.

1. W3C PROV data model with added integrity verification overloads the meaning of invalidity. An invalid bundle may denote that integrity of the bundle cannot be verified. On the other hand, an invalid bundle in W3C PROV signify that bundle does not meet requirements defined in W3C PROV constrains document [6]. Validity requirements specified within the standard are not part of this thesis. Therefore, invalidity denotes the inability to verify the integrity of a bundle.

3. Low-credibility bundles are bundles with an invalid bundle somewhere on a path from the initial bundle (included) to this bundle. The integrity of the bundle itself is verifiable. Therefore this category is more trustworthy than invalid bundles.

3.3.2 Integrity of meta-bundle

Meta-provenance contains the versioning history of bundles and metadata required to verify their integrity. Nevertheless, meta-provenance does not have integrity verification itself. For that reason, without any additional protection, a provenance in a meta-bundle can be modified without any detection. That is a problem because alternations of update history captured in meta-bundle can make the search algorithm use older or invalid versions of a bundle, for example, by removing a record of a newer version or vice versa by adding a record that captures a newer invalid bundle.

A solution to prevent meta-bundle modification can be to create a token for it. But this method is more computationally intensive, creates an additional layer of meta-provenance (meta-bundle versioning), stacks verification into multiple layers, and adds more data to maintain. Therefore this solution seems to be impractical. Another solution can be storing meta-bundle in a secure environment, where modification by an attacker is infeasible. When this is not possible, then there is a method that minimises the impact of meta-bundle alternations and consequently increases the reliability of meta-provenance. This method inserts confirmation about the older version in the bundle's newer version during the update before creating a token. The confirmation is revision-of relation stored in the newer version. This relation captures this bundle and its latest older version(Revision-of[ID of this version, ID of older version]). During verification of the newer version, the search algorithm also checks if this revision in the newer version complies with the update record in the meta-bundle. This method should be sufficient to prove that a specific bundle is indeed an updated version because a token in the meta-bundle protects the newer bundle. Hence, modifications of the updated bundle are detected together with added relation. However, if the version referenced by the `has_provenance` attribute is old and the newer version is invalid. There is no possible way to verify whether the concrete

bundle is the newer version of the referenced bundle. Therefore, the algorithm puts newer versions of this invalid bundle on the output with low credibility. More complex cases will be described in detail in chapter 4.

3.3.3 Search modes

There can be a dilemma which bundles should algorithm prefer. Whether the newest version without considering integrity or an older version with a valid token. Therefore there should be a possibility to tell algorithm preference between these situations. Two search modes are defined to address this issue. Each mode of the algorithm can significantly impact the output based on the validity of bundles.

1. Strict mode prefers a valid bundle over a newer version. The latest valid bundle can have different content and entities than the latest invalid bundle. Although, valid bundles are deemed more reliable than invalid ones.

2. Normal mode prefers the newest bundle even when it is invalid. Hence, the search in this mode is not affected by invalid bundles.

4 Possible cases of extended search algorithm

This chapter describes by illustrative cases possible situations in searching for an entity's related provenance in distributed provenance chains with extended updates and integrity verification. Some cases will show obstacles in search and result in required extensions for the general search algorithm 3.2.2.

4.1 Semantics and simplification of graphs

This section simplifies graphs that are later used to show different cases. Graphs without simplification of updates are difficult to read and quite complicated. Therefore, graphs will only show updates of bundles without specific relations like revision and specialization-Of and the meta-bundle in which they are stored. Furthermore, graphs attempt to abstract from the content of bundles where feasible and use colours to indicate the bundle's properties relevant for graph interpretation.

4.1.1 Simplification of graphs

Figure 4.1 shows how `has_provenance`, simple update, merge, and fork will be represented and simplified in graphs describing individual cases. The figure shows both types of graphs, one with content abstraction and the other where bundles keep entities with their `has_provenance` references and their derivations.

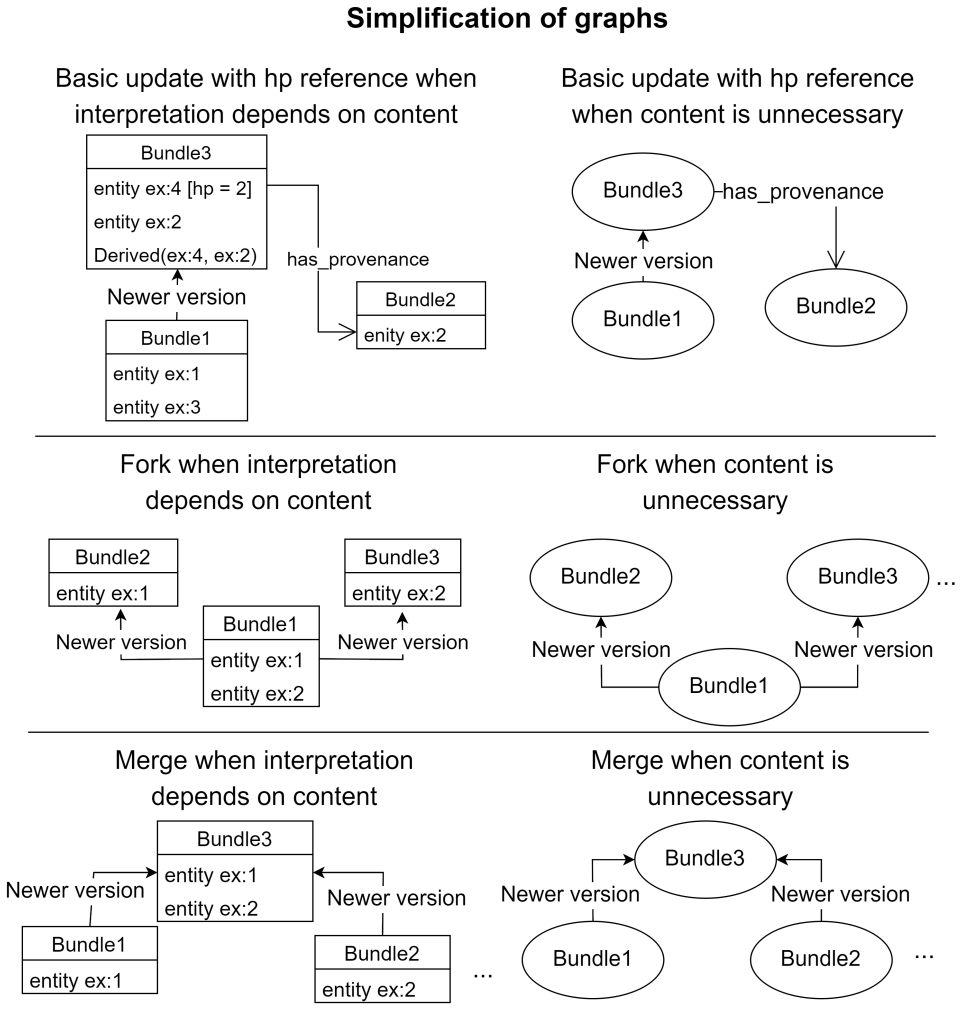

Figure 4.1: Simplification of relations in graph

Derived in the graphs represents a relation WasDerivedFrom [1] where a generated entity (in the Figure 4.1 entity ex:4) is in the first position and a used entity in the second position (entity ex:2). Hp is abbreviation for has_provenance.

4.1.2 Colours of bundles

Colours represent the presence of a searched entity or a referenced entity and the validity of a bundle's token. The figure 4.2 express each colour with details described below the figure.

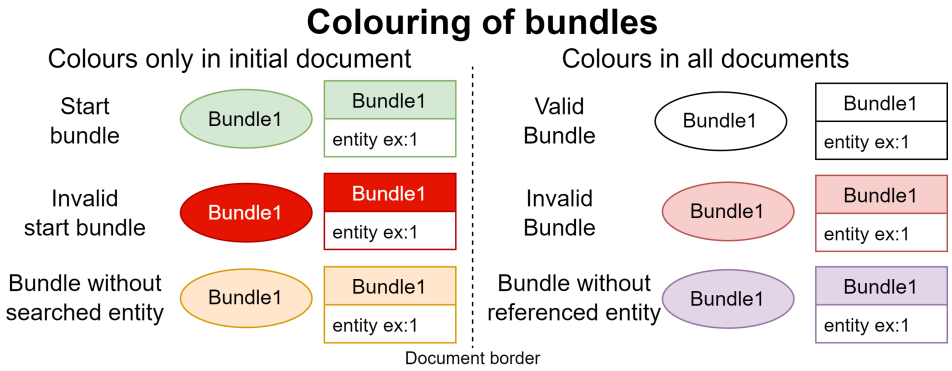

Figure 4.2: Meaning of bundle's colours within graphs

1. Start bundle represents the initial bundle in the initial document, the newest valid bundle containing the searched entity in the update branch.
2. Invalid start bundle is the same as the start bundle with the difference it has an unverifiable or missing token.
3. Bundle without searched entity is a bundle that does not contain searched entity in the initial document.
4. Valid bundle is a bundle with a verifiable and valid token.
5. Invalid bundle is a bundle with an invalid or missing token.
6. Bundle without referenced entity is a bundle that does not contain an entity that has referenced to this bundle by has_provenance.

A dashed separation line is used in figures to separate the content of individual documents. The most left part with bundles always represents the initial document. Validity does not matter in bundles without referenced or search entities because the search algorithm never includes or examines bundles without these entities. Therefore it is unnecessary to distinguish them by colours. Moreover, there can be one invalid start bundle and one valid start bundle in a single update branch due to the search modes that can influence the choice of an initial bundle.

4.2 Cases without updates

This section describes cases where the algorithm only traverses through `has_provenance` references with integrity verification.

4.2.1 Basic cases

The following enumeration describes the most basic cases in an environment without updates, integrity verification and backward references into already searched bundles.

1. The most straightforward case (depicted in Figure 4.3) shows only the initial document and its bundles without `has_provenance` references. Bundles containing the searched entity are initial bundles (where traversing will start). However, there are no links to other bundles, and therefore these bundles are the only ones that are traceable containing related provenance to the searched entity.

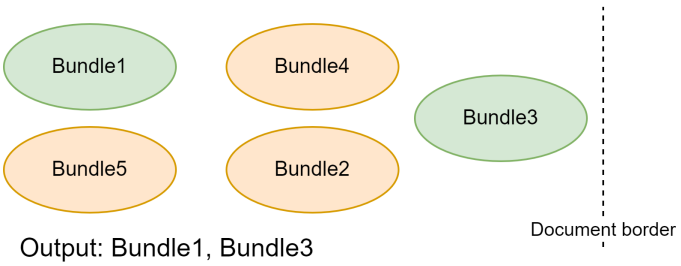

Figure 4.3: Basic case without has provenance

2. This case describes a setting where bundles contain at most one `has_provenance` reference, which cannot point into the initial bundle or any bundle that has been already referenced. `Has_provenance` may be transitive. Therefore, the algorithm must examine a bundle referenced from another bundle for further references. In Figure 4.4, the initial bundles are Bundle1 and Bundle3. All bundles transitively referenced from these bundles are in the algorithm output. References can also return to already searched documents where it can add extra bundles (an example is Bundle2).

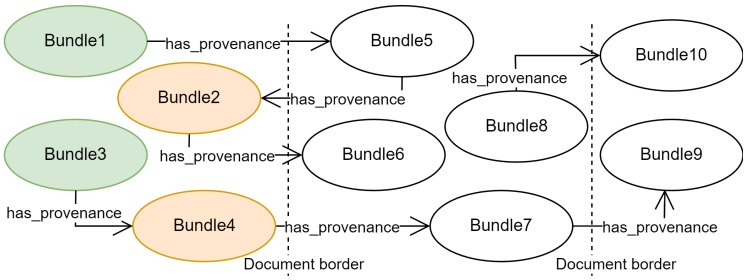

Figure 4.4: Basic case without interconnections of chains

3. This case shows the identical situation as the previous one, but this time describes how the content of bundles may affect the search. There can be five separate cases depicted in the following enumeration and Figure 4.5. Figure describing cases with content also shows the output with the searched (always entity ex:1) or referenced entities within each bundle to help a user determine which entities represent related objects to the searched object within a bundle.
   (a) Entity with `has_provenance` reference is not derived from the searched or referenced entity. Entity ex:3 within Bundle1 has not been used to generate the object represented by searched entity ex:1 because it is not captured by derivation. Therefore, entity ex:3 is unrelated to the searched entity and is not contained in the output together with all referenced bundles.
   (b) Referenced entity (an entity with ID which referenced into a bundle) or searched entity contains reference into a next bundle directly. Bundle2 contains searched entity ex:1 with a direct link to Bundle6, which also contains a direct link to Bundle8.
   (c) Reference can be in an entity that was derived from searched or referenced entity. Bundle4 has an entity ex:3 which is derived from the searched entity and references to Bundle5.
   (d) Reference(`Has_provenance` attribute) may have an invalid syntax, the referenced bundle does not exist, or some other problem occurred while acquiring a particular bundle. Bun-

dle5 contains a reference to a bundle that does not exist and also a reference with the wrong syntax. The algorithm should notify a user about these circumstances.

(e) Bundle does not contain referenced entity. For example, Bundle8 does not have an entity with ID ex:1 but only with ID ex:2. Therefore, the search algorithm does not add this bundle into the output because it does not contain relevant provenance or is unsure which entity represents the referenced entity. There should be some notification or warning to a user about this situation.

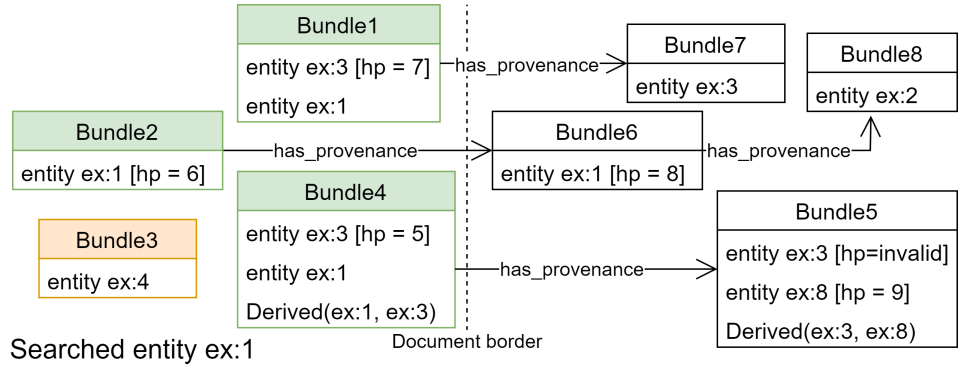

Figure 4.5: Basic case with content and no interconnections

4. Case where are no specific restrictions on linear `has_provenance` references (without cycles). There may occur a situation where the search algorithm finds a bundle that has been already found from different paths. For example in Figure 4.6, Bundle7 can be found by two paths from Bundle1: (1 -> 2 -> 4 -> 7) and (1 -> 6 -> 7). Furthermore, the search algorithm may find identical bundles from the different initial bundles, such as Bundle5, which can be discovered from Bundle1 and Bundle3, or Bundle2, which is referenced from the initial bundle with ID 1).

   Searching bundle multiple times can result in increased complexity and duplication of results. Therefore, a method that prevents the algorithm from traversing through the same bundle multiple times is required. The intuitive resolution is to remember already

discovered bundles and check whether the algorithm has not already processed the bundle after traversing to it.

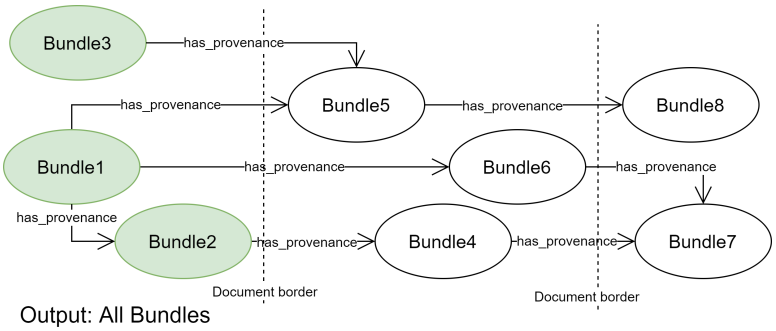

Figure 4.6: Basic case with interconnections

5. This case shows an identical setting as the previous one but describes how entities, their relations, and attributes(`has_provenance`) may influence the search. Figure 4.7 shows these situations on the graph, and the enumeration below explains every situation.

   (a) Bundle may have two or more entities with one or more references pointing into different bundles. Such as Bundle5 and Bundle1.

   (b) Bundle may have multiple entities that refer to the same bundle. These references point to different entities in other bundles that can further reference new bundles.
   This situation can be seen in Bundle3, which contains two entities pointing at Bundle5. Yet, entities ex:1 and ex:2 are not connected by derivation in Bundle5. Hence if only entity ex:2 had been examined from Bundle3, then Bundle7 would not have been discovered. Therefore, the algorithm should follow references from different entities to the already found bundle. However, it should check whether an entity with this ID has not been already examined in this bundle to prevent searching the bundle multiple times with the same entity. In other words, the algorithm should not only save IDs of searched bundles but also IDs of already examined entities within these bundles.

(c) Different entities from separate bundles can refer to the same bundle. The only difference from the previous case is that these references come from different bundles. Examples are Bundle6 which is referenced from Bundle1 and Bundle2 by separate entities, another example is Bundle9, but this time an entity referenced by `has_provenance` from Bundle8 is not present.

(d) Bundle may have an entity with several references to the same bundle. The algorithm must be prepared for these situations and not follow these references multiple times. Entity ex:4 in Bundle1 is a demonstration of this case.

(e) Bundle can have a transitive derivation of searched or referenced entity. An example is Bundle3, where searched entity ex:1 used entity ex:2, and before that entity ex:2 used entity ex:4. The algorithm should be able to traverse across has_provenance references from transitively derived entities.

(f) Bundle can have entities with the same ID, containing different `has_provenance` references. This behaviour is intentional in the data model because it allows quick attaching of new properties to a described object [1]. Therefore algorithm should traverse through all references in separate entities with identical IDs. An example is Bundle8, where two entities with ID ex:1 contain different references.

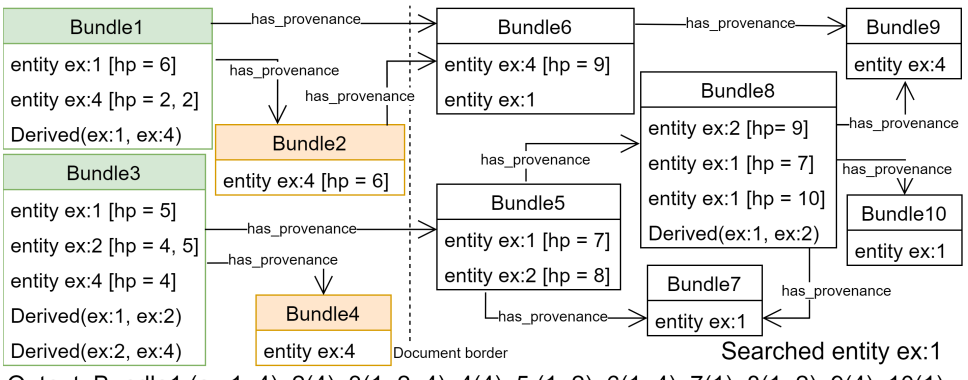

Figure 4.7: Basic case with interconnections and content

4.2.2 Cases with backward references

The following cases show situations that contain a cycle formed by backward `has_provenance` references into already discovered bundles.

1. Provenance chain may contain backward reference into an already searched bundle, resulting in an infinite loop when the algorithm does not expect this. The algorithm must be able to detect cycles to avert an endless loop. The algorithm checks whether it previously examined the bundle with a specific entity due to situations described in the previous case 5b. As a result, the algorithm can already prevent an infinite loop.

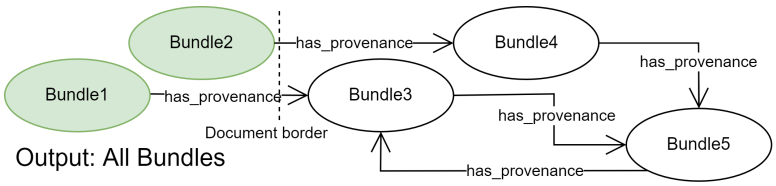

Figure 4.8: Has provenance cycle

2. Special case of a cycle is a bundle with an entity that contains attribute reference to the identical bundle.

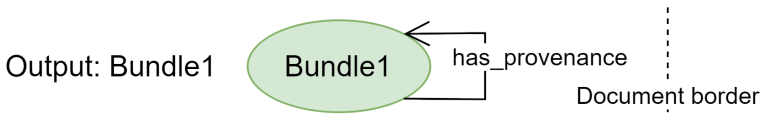

Figure 4.9: Special case of cycle

3. Backward reference does not have to necessarily result in a loop when a bundle was examined with a different entity(entities) than the entity with backward reference. Figure 4.10 shows backward reference by entity ex:3 from Bundle3, which does not create a loop and discover a new bundle with ID 4(Bundle4), but backward reference from Bundle3 by entity ex:1 creates a cycle.

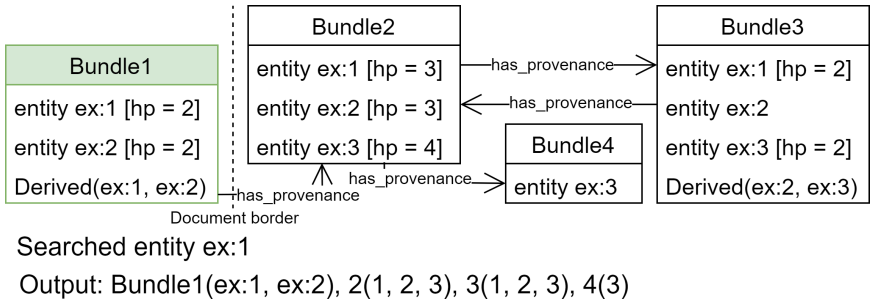

Figure 4.10: Has provenance cycle with content

4.2.3 Cases with integrity validation

Integrity validation separates the algorithm's output into three categories defined earlier(Subsection 3.3.1). These cases demonstrate how bundles are distributed into these categories by concrete examples in an environment without updates.

1. This case describes a situation where `has_provenance` references are linear, and every bundle is referenced at most from one other bundle. The case only shows that bundles (Bundle4, Bundle6, Bundle8) referenced after an invalid bundle in the path are in the output with low-credibility as defined earlier(Subsection 3.3.1).

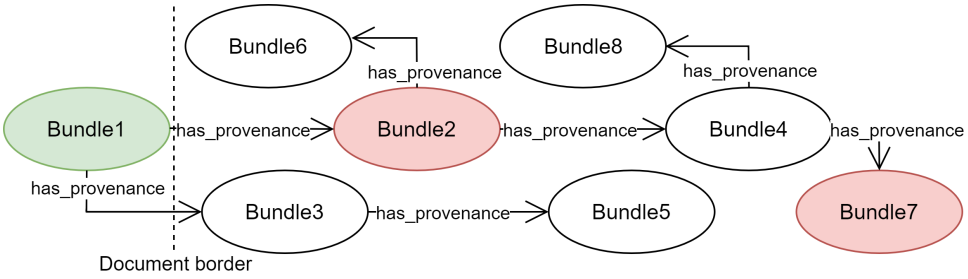

Figure 4.11: Integrity without interconnections of chains

2. Case with integrity validation and linear `has_provenance` references (no cycles). In contrast to the previous case, two or more paths may unite into a single bundle. When at least one of these paths is completely valid (does not contain any invalid bundle), the referenced bundle is also valid. Figure 4.12 shows that Bundle5 is referenced by completely valid path (1 -> 3 -> 5), therefore it is categorized as valid bundle on the output, even when another path (1 -> 2 -> 5) goes through invalid bundle.

   The algorithm can achieve this behaviour by traversing only through references within valid bundles until any left. After that, continue traversing references from invalid bundles.

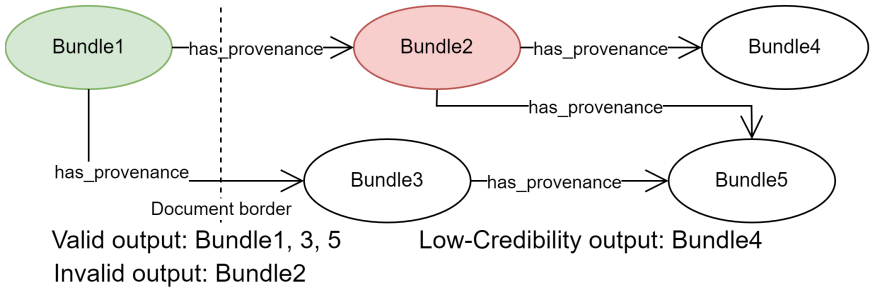

Figure 4.12: Integrity with interconnections of chains

3. This case shows how content influences the result of the algorithm with integrity verification in an environment with backward references. Entities which are separate(not mutually connected by derivation) and referenced from a path after an invalid bundle in an already found bundle can refer to new bundles. However, these bundles are marked as low-credible because they could not have been discovered from valid bundles.

   More explanatory is Figure 4.13 where Bundle5 is referenced from Bundle3 by a separate entity ex:2 findable only from invalid initial bundle with ID 2. Therefore, entity ex:2 in Bundle5 is in the output with low-credibility. Backward reference from Bundle4 after invalid Bundle6 finds new entity ex:3 within Bundle3. Hence entity ex:3 in Bundle3 is also on the output with low-credibility.

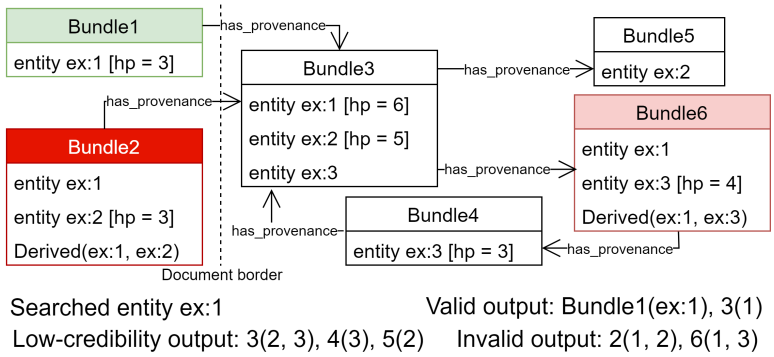

Figure 4.13: Integrity with interconnections and explicit content

4.3 Linear updates

This section describes situations while searching for relevant provenance in distributed provenance chains with linear updates.

4.3.1 Basic linear updates

Cases that represent the most elementary updates without integrity verification and references into already searched update branches.

1. Case where bundle may have newer versions. When the algorithm follows a reference to the older version, it must be able to detect and use more recent versions.

   The algorithm checks whether a bundle has a newer version in the meta-bundle by the current design of the update mechanism [3].

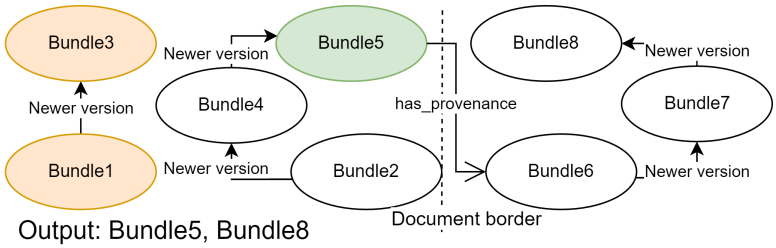

Figure 4.14: Basic linear update

Figure 4.14 shows the acquisition of the bundle's newer versions in the initial document (Bundle2) and the bundle in a foreign document(Bundle6) that got referenced into an older version. In both situations, the algorithm traverse versions by records in a meta-bundle until it finds the latest version of a particular bundle, from which the algorithm follows `has_provenance` from related entities. There can be update branches in the initial bundle that do not contain searched entity in any bundle (Bundle1 and Bundle3). These bundles do not occur on the output.

2. This case shows a situation where the newest version in an update chain may not contain searched entity while the older version does. The algorithm should use the latest version with the entity while informing a user when there is a newer version without a specified entity. Figure 4.15 shows this case in the initial and a foreign document. Bundle3 does not contain searched entity in the initial document, and Bundle7 does not contain referenced entity ex:2. Therefore, on the output are Bundle2 and Bundle6.

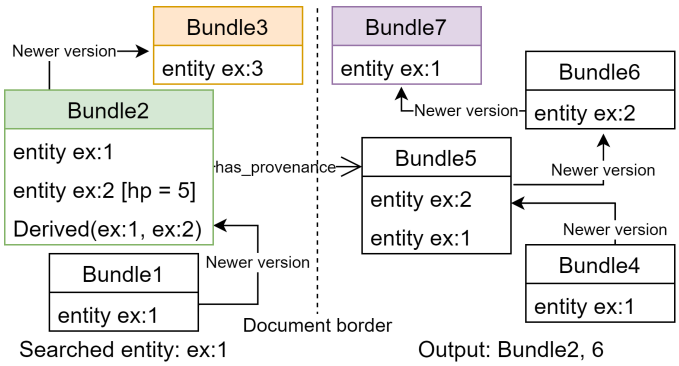

Figure 4.15: Basic linear update without entity in newer version

3. Case where an entity may contain a reference that points to a bundle without referenced entity. In Figure 4.16 entity ex:1 within Bundle1 refers to Bundle3 without this entity, not even in any of its newer versions. Older versions contain these entities. Nevertheless, they are not considered because there is no reason to use older versions of a referenced bundle. Bundle1 also has an entity linked to Bundle6, which does not have this entity, but

some newer versions do. Therefore, the latest ID of a bundle with a referenced entity occurs on the output. The algorithm should notify a user when a bundle and its newer version do not contain a referenced or searched entity.

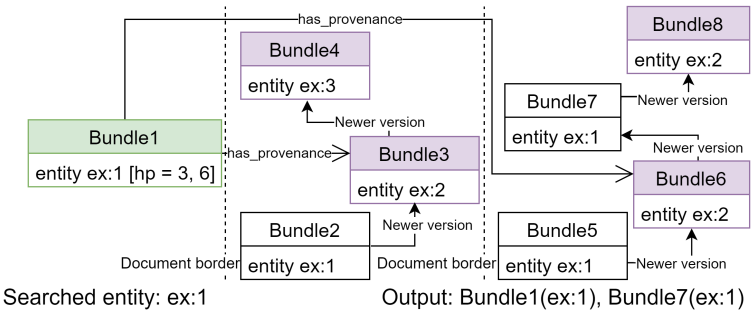

Figure 4.16: Basic linear update without entity

4.3.2 Linear updates with backward references

This section describes situations in updates where `has_provenance` can refer from a different initial bundle or by backward reference into a bundle within an already searched update branch.

1. This case shows a backward reference to an older version. As a result, this reference form a `has_provenance` cycle with newer versions. The algorithm should detect this reference and prevent loops, duplicates and unnecessary traversing to newer versions.

   For this reason, when the algorithm finds the latest version with searched or referenced entity, it adds all versions within its update branch with a given entity into already searched bundles.

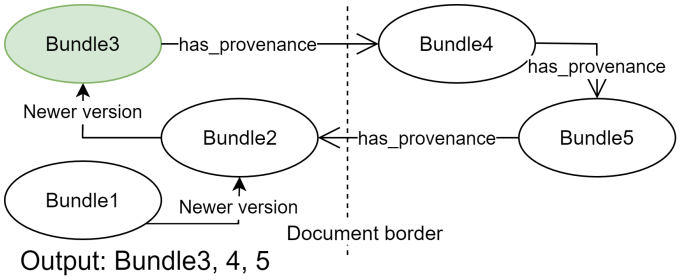

Figure 4.17: Backward reference into older version

2. This case describes a backward reference from an entity not examined in the update branch. This reference may find other entities or a newer bundle within the update branch. Figure 4.18 starts in Bundle3 with entity ex:1 marking all predecessors(older versions) as searched. Then the algorithm continues to Bundle5, where it finds entities with backward reference, ex:1 is already found in the referenced update branch, and entity ex:2 finds a new bundle with ID 4(Bundle4).

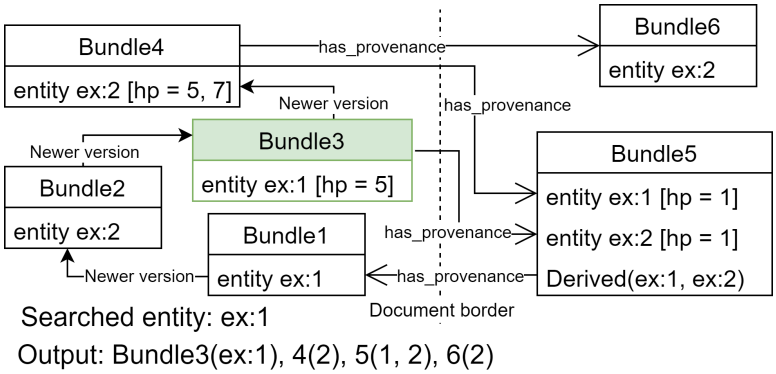

Figure 4.18: Backward reference into older version with different entity

3. This case describes a situation where an entity utilized in the generation of referenced entity is in a newer version while a referenced entity itself is not. This situation can cause different results and nondeterministic behaviour(explained later 3). Therefore the algorithm should check whether any newer version contains an entity used in the derivation of a referenced entity and use this bundle to traverse further references while marking all bundles in update branch as searched with this entity.

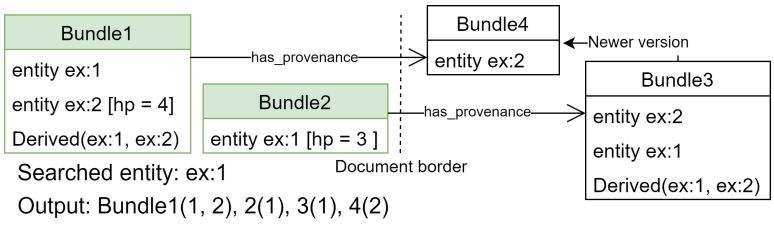

Figure 4.19: Linear update with reference into newer version

Figure 4.19 may indicate what can go wrong if this practice is not used. When the algorithm starts a search in Bundle1, it traverses to Bundle4, where it only finds entity ex:2. Then the algorithm marks predecessors of Bundle4 as searched with entity ex:2. The algorithm then examines the initial Bundle2 and finds references from the searched entity to Bundle3. In Bundle3, the algorithm adds into the output only entity ex:1 because ex:2 has already been marked as found by the algorithm. However, when the algorithm starts in Bundle2, it finds Bundle3 by reference from entity ex:1. Entity ex:1 utilized entity ex:2, therefore entity ex:2 is also added into result. Then the algorithm examines the initial bundle with ID 1 and follows reference from entity ex:2 into Bundle4, finding entity ex:2, but this entity was marked as searched from the older bundle. Therefore algorithm does not add Bundle4 with entity ex:2 into the result. Consequently, the result differs and depends on the choice of the execution order of initial bundles. Finding newer versions of derived entities solves this problem.

4.3.3 Linear update with integrity verification

This section is about possible situations during search with linear updates and integrity verification.

1. This case shows that the initial bundle or bundle referenced from a different bundle has a valid latest version, and some of its predecessors are invalid. In this situation, a user should be notified that there is an older version with an invalid signature, and the origin of the bundle is not entirely verifiable. The algorithm does not add these bundles into low credibility bundles because the bundle itself is valid and discoverable from a valid path.

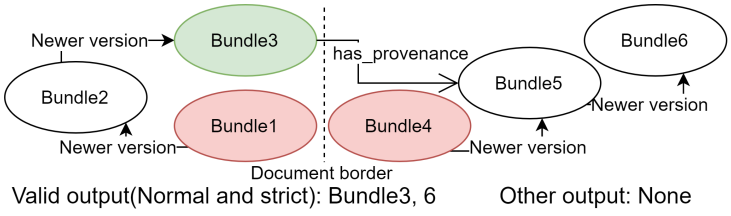

Figure 4.20: Invalid older version in update branch

2. Invalid bundle occurs in a newer version of a bundle referenced by `has_provenance` during the search. The algorithm uses the mechanism defined above(3.3.2) to confirm whether a specific bundle is indeed a newer version of some older bundle. However, this mechanism cannot be used when the algorithm traverse by update records within the meta-bundle into a newer version with an invalid token because confirmation in the newer version could have been changed. Therefore more recent valid versions of an invalid bundle are considered low-credible because the algorithm may have been redirected by record modification in the meta-bundle.

   There are two search modes defined earlier, strict and normal (section 3.3.3). The strict mode examines only the latest version with a valid token. Consequently, an invalid bundle cannot occur in the output of the strict mode. The normal mode takes the newest version with an entity without considering integrity.

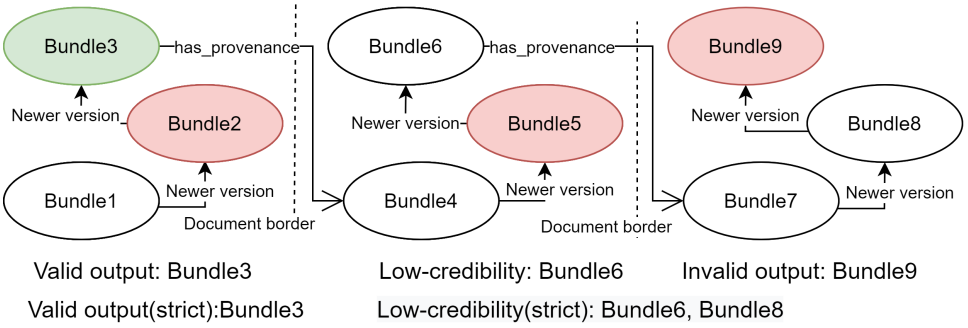

Figure 4.21: Invalid between older and newer version

Figure 4.21 shows the result of the strict and normal search for comparison. The algorithm has direct access to bundles in the initial document and does not have to cross through revision relations in meta-bundle. As a result, Bundle3 is in the valid output. Even it has an invalid older version.

3. This case describes the situation where `has_provenance` refers to an invalid bundle, but the newer version of this bundle is valid. The algorithm can verify that it is truly an updated version of

the referenced bundle due to the method defined in 3.3.2. When a newer bundle is not valid, it is not verifiable, and valid newer versions will be in low-credibility as in the previous case.

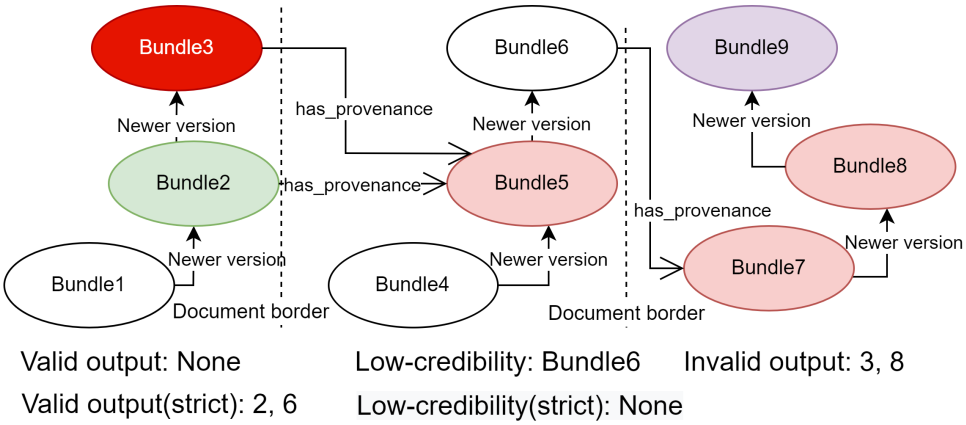

Figure 4.22: Reference into invalid bundle with update

Figure 4.22 illustrates that there can be different valid and invalid initial bundles in a single update branch depending on the selected mode (Bundle2 and Bundle3). Also, this figure shows that the algorithm in strict mode does not have to find a suitable bundle in referenced update branch, while normal does (reference from Bundle6). As a result, it does not add any bundle from this update branch to the output. On normal mode, it adds the latest bundle with the referenced entity (Bundle8).

4. This case demonstrates `has_provenance` reference from a valid path that points into a bundle, while another reference from a different initial bundle contains a link to an older version of the previous bundle. Furthermore, this older version must traverse through an invalid bundle to get to the identical newer version. When the algorithm crosses an invalid bundle, it must stop searching and wait until all references from valid bundles have been examined to prevent different results when starting from different initial bundles. If the algorithm finds a newer version of an invalid bundle that stops its search, then it adds this bundle to searched bundles. Therefore algorithm can later

detect that a newer version has been already found and does not further examine invalid bundle or its newer versions. Strict and normal modes behave the same way in this case.

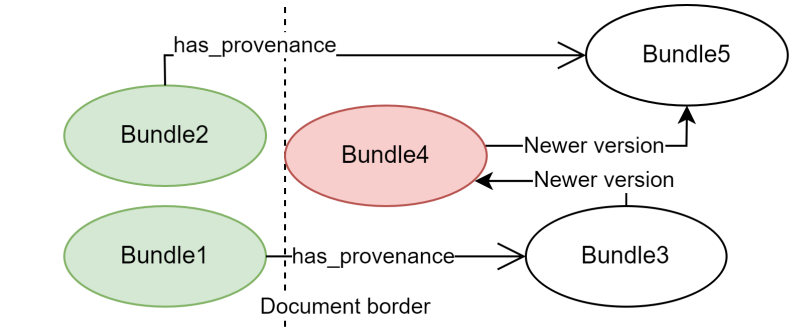

Figure 4.23: Multiple references into update branch with invalid bundle

4.4 Cases with merges and forks

This section describes new cases caused by merge and fork updates defined in section 2.3.

4.4.1 Merge cases

A structure created from the merge without has_provenance references can be imagined as a tree, where the newest bundle is a tree root. There are cases where a tree root is not included in the query result. For example, tree root is not valid during a search in strict mode, or the newest version does not contain searched entity, while some older versions do. Otherwise, a tree root is included in the output.

1. This case shows a merge with valid bundles. Merge can join two or more bundles. When the algorithm discovers the latest bundle containing the currently searched entity, it marks all predecessors as searched with this entity to prevent searching newer versions from multiple joined branches. In Figure 4.24 Bundle10 does not contain referenced entity. Therefore, the algorithm adds Bundle8 and Bundle9 into the result and marks their predecessors as searched because they are the latest version containing

entity. Bundle7 contains an entity but has not been transitively referenced from any initial bundle.

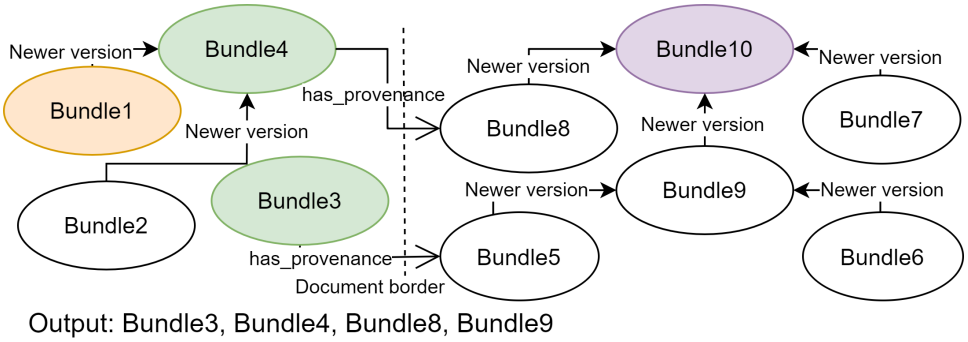

Figure 4.24: Merge of bundles

2. This case shows a merge with invalid bundles. Reference to an older bundle that must traverse through an invalid bundle should stop the algorithm from searching further as in linear update 4.23. There can be a backward reference to the update branch from an entity derived from a successor. Therefore derivations should also mark all predecessors as searched with derived entities (this also applies in linear updates). All other cases are similar to those without the merge.

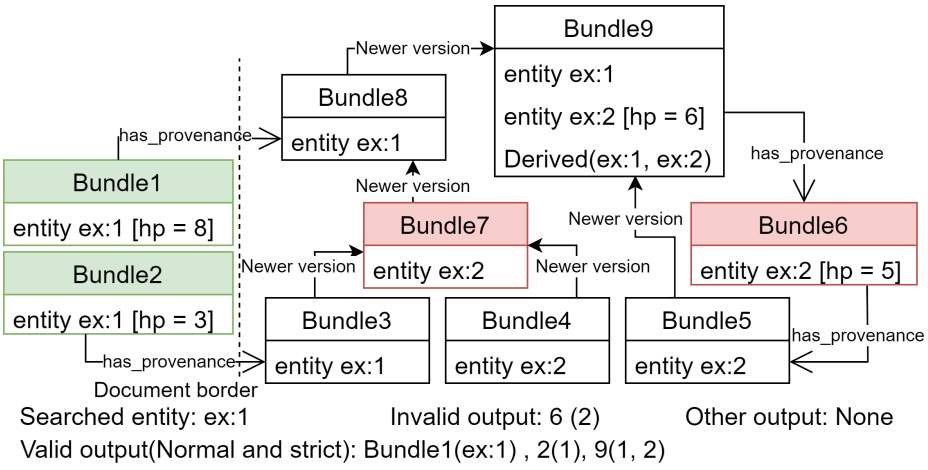

Figure 4.25: Merge with content and invalid bundles

4.4.2 Fork cases

This section describes cases with the fork which creates new update branches by splitting a content of a single bundle. These branches traverse through different bundles. Hence all branches should be searched for a particular entity.

1. This case shows the fork with invalid bundles, where the fork may split a bundle into several others. Bundles in an initial document or bundles referenced by `has_provenance` may have multiple branches with newer versions. Latest (valid if strict) versions with a searched or referenced entity within each update branch of this bundle are added to the result and used for further searching. Every update branch is searched separately. When a bundle contains searched or referenced entity while having newer branches that do not(furthermore are not valid in strict), then this bundle is added into the output with the warning that more recent versions do not have an entity like in linear update 2. However, when at least one branch contains the desired entity (in a valid bundle if strict), this bundle is not added, and the warning is not displayed. Figure 4.26 shows algorithm behaviour in a concrete example.

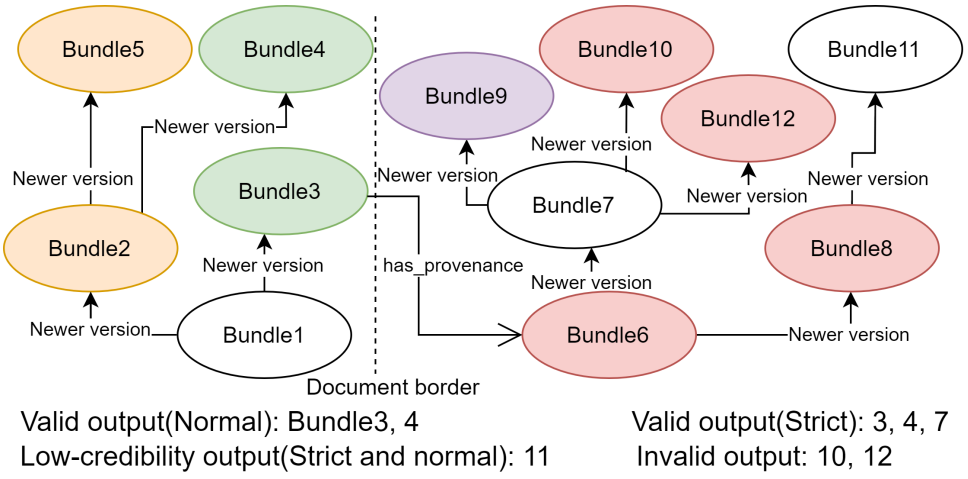

Figure 4.26: Fork of bundle with invalid bundles

2. This case describes a situation where the predecessor has multiple newer versions with backward references. Fork assigns multiple branches with newer versions to a single bundle. As a result, backward references can find new update branches when a referenced bundle has multiple of them created by the fork. Therefore, when adding predecessors into already searched bundles algorithm must check whether all newer versions have been searched with a concrete entity. If not, it cannot add this bundle to searched ones with all its predecessors. Otherwise, the algorithm would not find all bundles correctly in certain situations.

   For example, Bundle6 in Figure 4.27 would not have been found if all predecessors of Bundle10 were marked as searched with entity ex:1 without considering other newer versions.

   Furthermore, the fork adds a requirement to detect whether any newer version has searched or referenced entity to be able to show warning 1 or determine whether it needs to add this bundle into the output. A simple solution is to extend the data structure with searched bundles by information whether this bundle or any of its newer versions contain a searched or referenced entity.

   Integrity and combinations of merges and forks do not create any diverse situations that have not been discussed in previous cases.

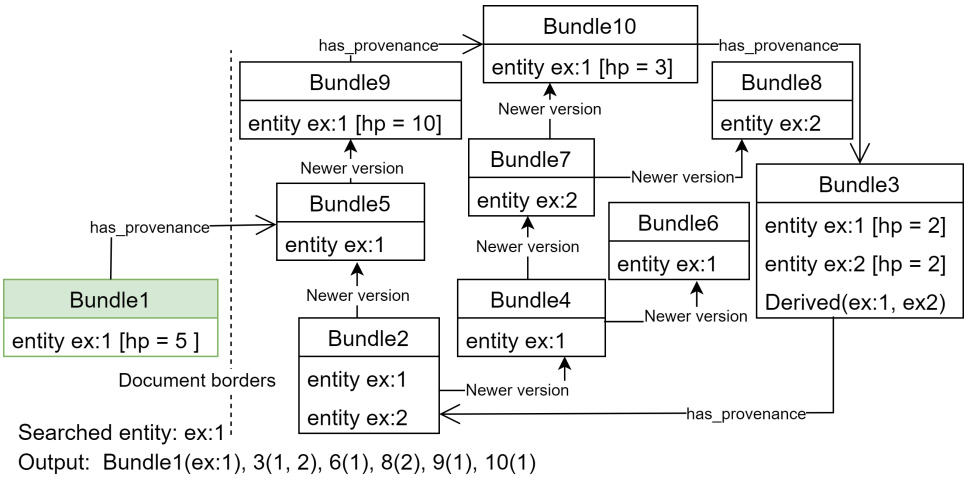

Figure 4.27: Fork with predecessors with multiple newer bundles

4.5 Updates in cycle

Updates in a cycle are not practical in the real world and are even forbidden by the W3C PROV standard (revisions in a cycle) [6]. Therefore, the algorithm should not process a document with updates in a cycle. It should instantly check the new meta-bundle for the occurrence of updates in a cycle. The algorithm should not continue processing the provenance document and display an error to a user if any update cycle is found. Moreover, the algorithm stops when the initial document contains an update loop because it cannot continue with the search.

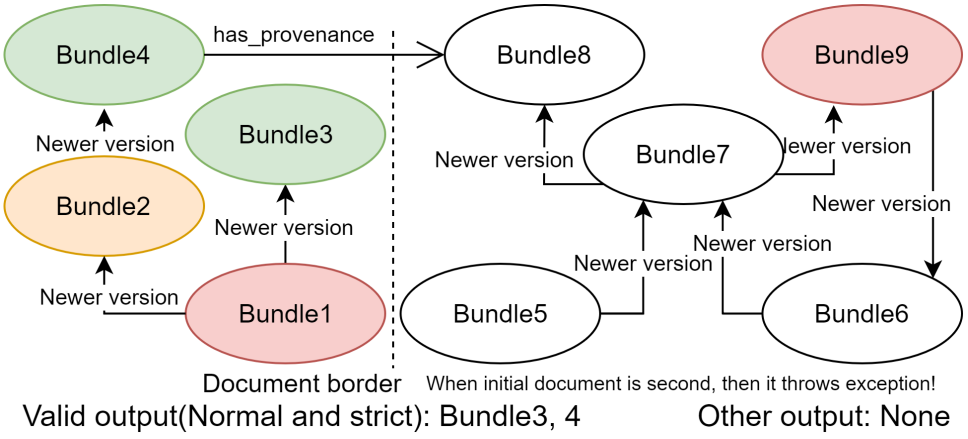

Figure 4.28: Document with updates in cycle

5 Implementation

This chapter depicts the structure of a prototype search algorithm in python and briefly summarizes how the algorithm works together with the generator that creates provenance documents with specific records in the W3C PROV data model. Additionally, this chapter cover details of the implementation of integrity verification.

5.1 Overview of the required functionality

This section briefly summarizes the most important observations of the required functionality to extend the general search algorithm 3.2.2 to enable searching an object's provenance represented in distributed chains with extended updates (linear, merge, fork and cycle updates) of its parts and their integrity verification.

1. The integrity verification is done by validation of records in bundle against digital signature stored in token within meta-bundle 3.1. It separates the output into three categories 3.3.1 and including updates adds the necessity to define at least two modes of the algorithm 3.3.3.
2. After the algorithm traverse to another by `has_provenance`, it checks its validity and seeks the more recent version by traversing revision relations in the meta-bundle 2.2. While crossing them, the algorithm checks confirmation stored in the newer version 3.3.2 to minimize the impact of meta-bundle modifications. More recent versions are also tested if their token complies with data stored within them.
3. The algorithm must hold visited bundles in provenance documents with their entities in a particular data structure. This data structure is called searched bundles and saves information if a combination of entity and bundle has been searched and whether the bundle or any newer versions holds this entity. Therefore, this data structure allows checking if any bundle has been examined and contains an entity to prevent cycle, unnecessary traversing, and duplicates.
4. The algorithm must check deserialized provenance documents for updates in a cycle. Updates in a loop make the whole docu-

ment faulty and untrustworthy. Therefore, the algorithm should not use this document for further searching 4.5. An initial document that contains updates in a cycle makes the algorithm unable to execute the search query 4.5.

5. The algorithm should notify a user when newer versions do not contain a specific entity, an invalid bundle occurred with an explanation of why it is invalid, and about issues that arise during the acquisition of referenced bundle. Issues include a missing entity, not existing bundle, invalid syntax of reference, document with updates in a cycle, missing referenced entity, and invalid token in the older version.
6. Bundle with an undiscovered entity that is referenced after an invalid bundle should be put in the output with low credibility because it may have been redirected from the truthful result. However, if there is an entirely valid path to the specific entity within the bundle, it cannot appear on the output with low credibility (only valid or invalid output). The algorithm can achieve this behaviour by traversing first through valid bundles and then continue following references from invalid ones.
7. Bundle added on the output and used for further searching should be marked with all newer versions and predecessors that have all newer versions examined with this entity as searched to prevent cycle and unnecessary traversing to more recent versions.
8. The algorithm should review updated versions to check whether they contain derivation of searched or referenced entity because it may occur there while the base entity does not. This practice prevents different results based on the processing order of initial bundles.
9. If the algorithm traverse through an invalid version to get to the valid newer version. Then it should treat this valid version with low credibility, including all transitively referenced bundles from entities within this bundle. The algorithm should wait until all other valid bundles have been searched and then verify that the invalid bundle has not been already included in searched bundles while it has been postponed. If not, the algorithm adds this bundle to the low credibility output and traverses its references.
10. The algorithm should treat newer branches created from the fork as separate. In other words, add the latest bundle with a specific

entity in each branch to the output and follow its references. However, when none of these update branches contains the (valid in strict mode) bundle with the entity, the forked bundle is added to the output with its transitive references 1.

5.2 Python library

The implementation uses a python library to capture provenance structures and serialize them into provenance documents in various formats. This library can serialize structures into PROV-JSON, PROV-XML or standard specific PROV-N notation [11]. However, deserialization is only possible from documents in XML and JSON. The W3C PROV standard documents how serialization looks like in each format [2].

5.3 Provenance generator

The cases described in the previous chapter 4 demonstrated various situations during the search by the algorithm for an object's related provenance. Some cases also illustrated the necessity to extend the general algorithm described in chapter 3.2.2.

I can use the previously defined cases to determine whether the final algorithm can correctly traverse and find provenance within distributed chains with updates and integrity verification. The provenance generator can create provenance documents by user specification. Therefore, I will use it to create documents that participate in cases within chapter 4. The algorithm then processes these documents and prints the output to compare it with theoretical results.

5.3.1 Generator functionality

The generator provides the functionality to create specific provenance documents according to the provenance data model(PROV-DM). These documents demonstrate that the proposed algorithm can handle various cases in distributed provenance chains. The generator has input, where the user can specify bundles with entities and their `has_provenance` references. Furthermore, the generator can make updates of bundles (linear, merge, fork and cycle) with specific changes

of records within updated bundles. Update relations are together with created tokens for bundles stored in meta-bundle.

After a generation of a bundle, the generator signs a hash of this bundle. Then it creates a token in the meta-bundle containing this signature and all necessary information to verify the bundle's integrity. Specification for hash function and sign function is set by a global variable stored in the Utilities module.

The generator does not realize the generation of any provenance document. Still, its functionality is sufficient for creating documents with records that influence the search to test the functionality of the implemented prototype of the search algorithm.

Figure 5.1 shows a simplified activity diagram of the generator

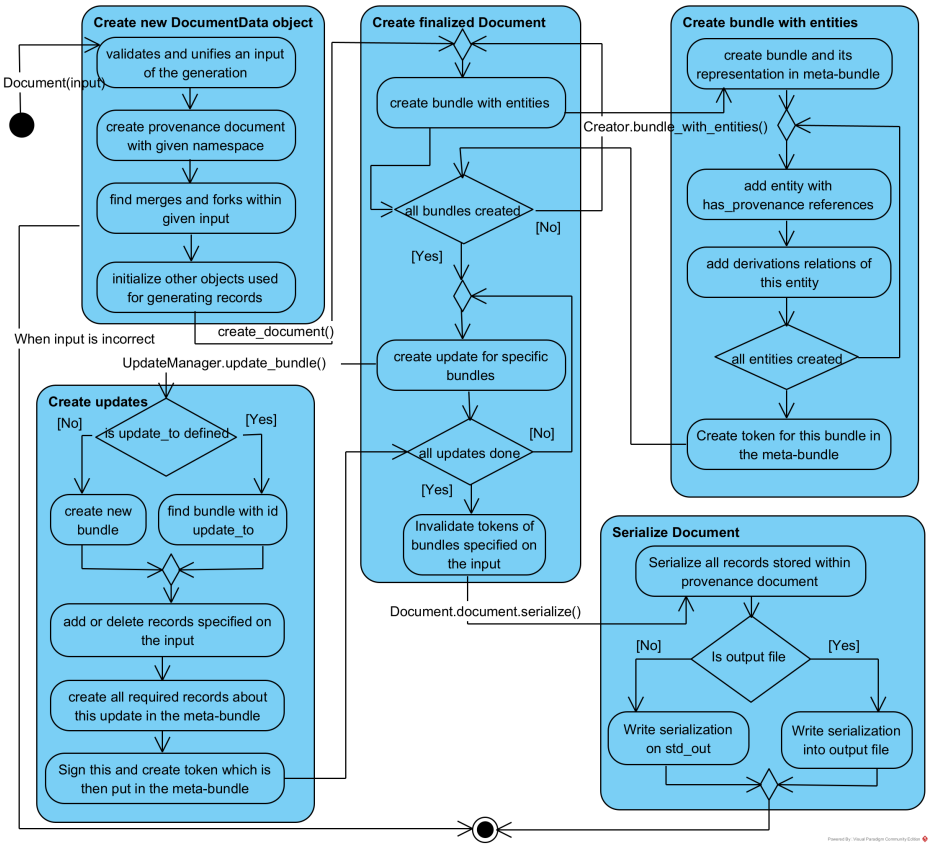

Figure 5.1: Generator activity diagram

5.3.2 Generator input

Generator input consists of five parameters: bundles, updates, file_path, invalidate_bundles, and start_id. The enumeration below explains each parameter.

1. `Bundles_with_entities`: This parameter defines bundles in the resulting provenance document with their entities. The entity represented in the generator must contain its ID and may have defined `has_provenance` references into a bundle in a specific format[1]. An entity can have also defined other entities within a bundle from which it is derived. A more precise description of this parameter is within the implementation of the generator.

2. `Updates`: This parameter defines updates of specific bundles. It can be used to specify which bundles will be updated and how the update changes the bundle's content by explicit deletion of entities or addition of new entities with specifications of their ID, has_provenance references and entities from which it has been derived. An update does not have to create a new bundle but can use an already created one to allow the merge of bundles and update loops. A more detailed description of this parameter is in the implementation of the generator as well.

3. `File_path`: This parameter specifies the path in python-format where the generator will save a serialized provenance document. When this parameter cannot be used as the path with sufficient privileges, it warns the user and prints the result of serialization on the std-out (standard output) instead of the specified file. If the path is not specified, it only prints serialization on the std-out without notifying a user.

4. `Invalidate_bundles`: This parameter represents IDs of bundles for which the generator will invalidate token by modification of a bundle after its generation.

5. `Start_id`: This parameter states the first number representing the ID of the bundle at the first position specified in the parameter

1. Path in python-format (https://docs.python.org/3/library/pathlib.html) of the document/ID of a bundle within this document.

`bundles_with_entities`, from which numbering of the other IDs continue by their position.

5.4 Integrity verification

The way how the integrity verification is designed within distributed provenance chains has been already defined in chapter 3.1. This section describes only details such as how to hash bundles with their content and essential information necessary to verify integrity.

5.4.1 Hashing of bundles

Currently, the python library does not implement serialization of bundles but only documents. To be able to hash bundle, it must be encoded in bytes. Therefore, I propose the conversion method of a bundle with its records into a unique string that will be transferred to bytes in the specified encoding. This transformation does not have to be reversible.

Deserialization of documents does not have to preserve the order of individual records and their attributes. As a result, attributes and records within a bundle must be uniformly ordered before using signing and validation functions to prevent different hashes depending on the arrangement of records.

5.4.2 Content of token

It is clear that a token needs to contain a digital signature of a particular bundle's hash. Although, the verification procedure requires more information to verify the integrity of the bundle. Verification is not realizable without knowledge of a public key and used hash and sign function. This information is usually packed in a certificate of a public key. Still, the implementation is simplified and packs all required information in a token without validation against a certificate issued by a trusted certification authority.

The token contains encoding (used on strings representing bundle's content before hashing), expiration date, identification of hash function, a hash of bundle, a public key, a signature of the bundle, specification of sign function, time of a token creation in both signed and plain forms.

Each bundle within documents is signed separately, and every token can contain a different public key. Therefore individual bundles can be verified and signed by different public-secret key pairs.

5.5 Structure of the implementation

The program has six modules maintaining different functionality. This section describes each module and its purpose.

5.5.1 Search

Search is the main module for the entire implementation. It packs the logic behind seeking related provenance of an entity in an environment with extended updates and integrity verification. It can deserialize provenance documents and discover entities within bundles that represent objects that participated in creating the object represented by an entity passed on the algorithm's input.

5.5.2 Generator

The generator module implements the generator described previously 5.3.

5.5.3 Utilities

Utilities is the module that contain auxiliary functions used across other modules. These functions usually extract or find data from provenance structures or documents and create local files by a path. Utilities also contain global variables. These variables specify sign and hash functions used to create tokens, the format of the serialized provenance documents, endianity of integers representing sign and hash, encoding of strings before their hashing, days after which the token will expire, prefix with URI used within documents.

5.5.4 LinkedList

LinkedList is the module that implements the data structure linked list. Currently, in python is no official library implementing a linked list

data structure. Therefore I used this site [12] to create a linked list to increase the efficiency of removing specific values within a collection by this data structure.

5.5.5 Crypto

The crypto module implements functionality that allows verifying the integrity of a bundle. In detail, the module can be used to hash bundles, issue digital signatures for any information, create and pack tokens with data, generate new key pairs, parse and extract data from tokens and validate them against a specific bundle. The crypto module can use signature functions like elliptic curves and RSA with different byte lengths. Specification of used cryptography function is in global variables within the Utilities module.

5.5.6 Test

Test is the module which shows that implementation works as intended. It generates documents for all cases in chapter 4. This module then runs on these documents the search algorithm and prints results separated into three categories 3.3.1. The result contains a document, bundle and related entities to the searched entity within this bundle. Also, the output shows warning about situations that occurred during the search. The module also supports manual tests by the user, which can specify the initial document, mode and searched entity. The module then tries to run the search from these specifications and show the output of the search query.

The test module shows that the prototype search algorithm can search an object's provenance represented within distributed provenance chains with updates and integrity verification of its parts. Therefore, the implementation confirms that these features do not disturb the traceability of an object's provenance when the search algorithm adheres to the observations defined in this thesis.

6 Results and conclusion

This thesis aimed to analyze how updates and integrity verification of separate parts of distributed provenance chains affect the traceability of an object's provenance represented by these chains. More specifically, I studied the provenance standard (W3C PROV) and its data model and learned how distributed provenance chains and updates of their parts are currently realized. Furthermore, I proposed a way to achieve the integrity of provenance parts and worked out potential situations during the search for an object's provenance with updates and integrity verification. At last, I implemented the prototype of the search algorithm to verify that the updates and integrity verification do not disturb the traceability of an object's provenance.

At first, the thesis described the W3C PROV data model that unifies the provenance representation of objects in bundles, explained why provenance about a single object may be spread into diverse locations and depicted the means to interconnect bundles with an object's related provenance. Later on, it covered an existing mechanism for updating bundles.

I extended this mechanism to support the splitting and joining of bundles. Moreover, I defined how data necessary to verify integrity could be captured in meta-provenance together with records of updates. Subsequently, I proposed a simple general algorithm that searches for provenance without updates and integrity verification, explained how integrity may influence the search result, and defined why the integrity separates the output into three categories. After that, I expressed different cases in an environment with extended updates (linear updates, merges, forks and updates in cycle) and integrity verification that may occur during the search for bundles with related provenance of an object represented by the entity. These cases may have demonstrated obstacles during the search and proposed improvements for the general algorithm to resolve shown problems. As a result, the general algorithm has been gradually extended to address additional requirements that resulted from the analysis by cases.

Finally, I described the generator that can produce provenance documents with serialized records of all cases and address the requirements of the extended search algorithm. Documents created by the generator have been used to verify the prototype of the extended search algorithm, which demonstrated that it is possible to search an object's provenance. This algorithm proves that the search for related provenance can be achieved in distributed provenance chains with updates of its parts and their integrity verification by following specific rules established in the thesis.

Bibliography

1. *Provenance data model*. World Wide Web Consortium, 2013-04-30. Available also from: https://www.w3.org/TR/2013/REC-prov-dm-20130430/.
2. *Overview of standard*. World Wide Web Consortium, 2013-04-30. Available also from: https://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.
3. Building a digital space for the life sciences. 2021. Available also from: https://zenodo.org/record/4705074#.YiyZw0DMJPa.
4. *PROV-N notation*. World Wide Web Consortium, 2013-04-30. Available also from: https://www.w3.org/TR/2013/REC-prov-n-20130430/.
5. *Overview of provenance*. World Wide Web Consortium, 2013-04-30. Available also from: https://www.w3.org/TR/2013/NOTE-prov-primer-20130430/.
6. *Constrains in provenance*. World Wide Web Consortium, 2013-04-30. Available also from: https://www.w3.org/TR/2013/REC-prov-constraints-20130430/.
7. SAMUEL, Sheeba; KÖNIG-RIES, Birgitta. End-to-End provenance representation for the understandability and reproducibility of scientific experiments using a semantic approach. *Journal of Biomedical Semantics*. 2022, vol. 13, no. 1, p. 1. ISSN 2041-1480. Available from DOI: 10.1186/s13326-021-00253-1.
8. *Has provenance definition*. World Wide Web Consortium, 2013-04-30. Available also from: https://www.w3.org/TR/2013/NOTE-prov-aq-20130430/#resource-accessed-by-http.
9. *Handbook of applied cryptography*. Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, 2011-07-08. Available also from: https://cacr.uwaterloo.ca/hac/.

10. HASAN, Ragib; SION, Radu; WINSLETT, Marianne. *Proceedings of the 2007 ACM Workshop on Storage Security and Survivability*. Introducing Secure Provenance: Problems and Challenges. Alexandria, Virginia, USA: Association for Computing Machinery, 2007. StorageSS '07. ISBN 9781595938916. Available from DOI: 10.1145/1314313.1314318.

11. *W3C PROV python library introduction*. Trung Dong Huynh, 2021. Available also from: https://prov.readthedocs.io/en/latest/readme.html.

12. MYRIANTHOUS, Giorgos. How to Implement a Linked List in Python [online]. [N.d.] [visited on 2021-12-20]. Available from: https://towardsdatascience.com/python-linked-lists-c3622205da81.