Linköping Studies in Science and Technology
Thesis No. 1216

Understanding Certificate Revocation

Åsa Hagström

LIU-TEK-LIC-2006:1
Department of Electrical Engineering
Linköpings universitet, SE-581 83 Linköping, Sweden

Linköping, 2006

Understanding Certificate Revocation

© 2006 Åsa Hagström

Department of Electrical Engineering
Linköpings universitet, SE-581 83 Linköping, Sweden

ISBN 91-85457-84-1 ISSN 0280-7971

Printed in Sweden by LiU-Tryck, Linköping 2006

Hey...
What did you hear me say?
You know the difference it makes
What did you hear me say?
Yes, I said it's fine before
But I don't think so no more
I said it's fine before

I've changed my mind
I take it back
Erase and rewind
'Cause I've been changing my mind
Erase and rewind
'Cause I've been changing my mind
I've changed my mind

The Cardigans

Abstract

Correct certificate revocation practices are essential to each public-key infrastructure. While there exist a number of protocols to achieve revocation in PKI systems, there has been very little work on the theory behind it: Which different types of revocation can be identified? What is the intended effect of a specific revocation type to the knowledge base of each entity?

As a first step towards a methodology for the development of reliable models, we present a graph-based formalism for specification and reasoning about the distribution and revocation of public keys and certificates. The model is an abstract generalization of existing PKIs and distributed in nature; each entity can issue certificates for public keys that they have confidence in, and distribute or revoke these to and from other entities.

Each entity has its own public-key base and can derive new knowledge by combining this knowledge with certificates signed with known keys. Each statement that is deduced or quoted within the system derives its support from original knowledge formed outside the system. When such original knowledge is removed, all statements that depended upon it are removed as well. Cyclic support is avoided through the use of support sets.

We define different revocation reasons and show how they can be modelled as specific actions. Revocation by removal, by inactivation, and by negation are all included. By policy, negative statements are the strongest, and positive are the weakest. Collisions are avoided by removing the weaker statement and, when necessary, its support.

Graph transformation rules are the chosen formalism. Rules are either interactive changes that can be applied by entities, or automatically applied deductions that keep the system sound and complete after the application of an interactive rule.

We show that the proposed model is sound and complete with respect to our definition of a valid state.

Acknowledgements

The single most important person in helping me write this thesis has been Francesco Parisi-Presicce, professor at the Department of Information and Software Engineering of George Mason University in Fairfax, Virginia. He has provided many of the ideas presented herein, as well as graph theoretical expertise. Francesco — this work took a lot more time to conclude than you had bargained for, and I am eternally grateful that you stayed around to see it finished. Thank you for being my collaborator, my mentor and my friend!

I am grateful to the Swedish Fulbright Commission, the Hans Werthén Fund of the Royal Swedish Academy for Engineering Sciences, and the LM Ericsson Fund for Research in Electrical Engineering, for funding my visit to George Mason University during the year that provided the spark for this work. I am much obliged to Sushil Jajodia for hosting me at GMU, and to my colleagues at the Center for Secure Information Systems for welcoming me.

I would also like to thank my advisors Ingemar Ingemarsson and Robert Forchheimer for helping me see this work through, and all of my colleagues in the telecommunications corridor here at ISY for their company and friendship. My office roommate Kristin Anderson deserves a special mention for putting up with me, and for being such a patient listener.

My friend Jakob Heinemann helped me improve the cover design — thank you for your advice.

Finally, kudos to Patrik and to the rest of my family for always supporting my crazy endeavors!

Linköping, February 2006

Åsa

Contents

Chapter 1

Introduction

In this thesis we present a conceptual graph-based model to better understand the semantics of certificate revocation. The model describes the knowledge of all entities in a system simultaneously, and is distributed in nature. In other words, we allow every entity to issue, distribute and revoke its own certificates to and from others.

Our main purpose is to understand and clarify the concept of revocation in the context of a public-key infrastructure.

1.1 Public-Key Technology

Companies and organizations in today's world rely heavily on information systems to conduct their business. Crucial to these actors are security aspects that give confidence and legal validity to their transactions, for example:

- identity verification — establishing confidence in the identity of other parties;
- confidentiality of information — hiding sensitive information from all but the authorized parties;
- integrity of digital files — establishing confidence that information has not been tampered with;
- non-repudiation of contractual agreements — binding parties to their signature on a contract;

- time-stamping of transactions — knowing when a transaction took place.

A *public-key infrastructure*, or PKI, can supply all of these services to an information system through the use of *public-key cryptography*.

For more detailed information on public-key cryptography and PKI, see for example *Handbook of Applied Cryptography* [MvOV97] or *Understanding PKI: Concepts, Standards, and Deployment Considerations* [AL02].

1.1.1 Public-Key Cryptography

If Alice and Bob want to exchange secret information using classical symmetrical (or secret-key) cryptography, they each need a copy of a mutual secret key. If Bob also wants to exchange information with Carol, he needs to share another key with her, and so on. However, in public-key cryptography, there is no need for separate key pairs for each pair of collaborators.

In public-key cryptography, the key that encrypts a piece of information is not the same key that can decrypt it. Instead of sharing secret keys with their collaborators, each user has their own public-key pair. Such a key pair consists of one *private key* and one *public key*, created together by the use of a specific mathematical formula. The private key is kept secret by its owner, but the public key can be distributed to any user. The security of public-key cryptography rests on the mathematical difficulty in calculating one of the keys in a pair, given the other key. Examples of such intractable problems are factoring large integers, computing square roots in large integer fields or finding the discrete logarithm of elements of a cyclic group. This way only the person who has generated a key pair knows the private key — as long as it is kept secret, of course.

Information that is encrypted with an entity's public key can only be decrypted using the corresponding private key. Anyone who knows the public key of Bob can encrypt data for him, but only Bob — who knows the private key — can decrypt it. In other words, Alice and Carol can both use Bob's public key to send him information, but neither one can decrypt what the other has sent.

Assume that Bob has a key pair, $B.K$ and $B.k$ (we use a capital K to denote the public key and a lowercase k to denote the private key). If Alice wants to share some secret information S with Bob, she encrypts S using Bob's public key $B.K$, and transmits the encrypted data to Bob. When Bob receives it, he applies the decryption algorithm with his private key $B.k$, which produces S [1]. Since Bob is the only one with access to $B.k$, he is the only one who can decrypt the message. This is how confidentiality is achieved in public-key cryptography.

Digital Certificates

In order for the scheme described above to work, Alice must be convinced that $B.K$ is in fact Bob's key and not the key of some adversary Eve (if it were, Eve would be the one who could decrypt S instead!). This can be achieved through the use of *digital signatures*. When the decryption algorithm is applied to an unencrypted data piece using the private key, a digital signature is produced. By appending this signature to a message, anyone with knowledge of the corresponding public key can apply the encryption algorithm to the signature and check if the result equals the message that was signed.

One way to use signatures is to have a TTP (trusted third party) vouching for the authenticity of $B.K$. Assume that Alice has a properly verified copy of the TTP's public key, perhaps coded into Alice's hardware, or received through certified mail. The TTP can append a digital signature to a copy of $B.K$, vouching for its authenticity. This signed statement is what is known as a *digital certificate*. Since Alice trusts the TTP, verifying the signature on the certificate will give her confidence in the authenticity of $B.K$, as well as in the integrity of the information — if the signature can be verified, Alice will know that no one has tampered with the information since the signature was made.

Figure 1.1 shows the structure of a version 3 X.509 certificate, the most widely-used type of certificate. Most of the field names should be self-explanatory. The signature field indicates the algorithm used in the digital signature. The validity field specifies a time frame when the certificate is to be considered valid, unless it has been revoked. Possible extensions include authority and subject key identifier, key usage (i.e. what the key is to be used for, e.g. signatures, non-repudiation, key agreement etc), policy constraints, and more.

[1]In reality, encrypting and decrypting large amounts of data with public-key cryptography is computationally very slow. Therefore, actual encryption protocols use a combination of public-key and symmetric cryptography. In the context of this thesis, we are not concerned with that level of detail about the encryption particulars, so we will model the process as if it used public-key encryption only.

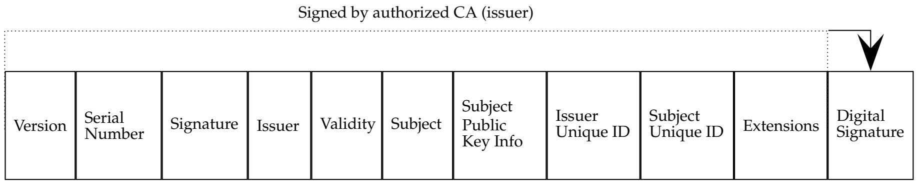


Figure 1.1. The structure of an X.509 version 3 certificate [AL02]

1.1.2 Identity

The concept of identity is inherently linked to digital certificates, because most certificate types bind a key to an entity's identity. However, capturing the identity of an entity in a way that is globally unique and meaningful to others is not a trivial task.

The X.509 standard [X50900] is based on the X.500 Distinguished Name (DN) structure [X50004]. The aim of the X.500 model is to assign every entity a global, unique identity, based on a hierarchical structure (e.g. country-company-unit-given name). The effect of an X.509 certificate is a binding between such a Distinguished Name and a public key. In practice, since there is no worldwide X.500 directory, deployers of X.509 PKIs often set up local (e.g. company-wide) X.500 directories.

The opponents of X.509-based PKIs argue that the creation of a worldwide directory is unlikely, and that a DN may be relevant in some contexts but not in others. For example, when the government employee Bob wants to contact Alice Smith in another department, the DN structure is obvious and well-known to him, but when Alice's old friend Carol wants to do the same, she has no idea which of all the possible Alice Smiths in that department to choose. Carol can obtain certificates for all possible Alice Smiths, but unless she can understand the DN structure they are of no use to her.

An alternative view of identity is to use local names. Each user then creates a local namespace with their own preferred nicknames of other entities, and issue certificates for these entities. The local name is the pair of a public key and this nickname (an arbitrary identifier) — thus, the identity of a user is represented by the public key(s) it controls. This is the view adopted by SPKI/SDSI. Ellison [CE96] presents different protocols to use for key exchange between entities, depending on the relationship between them. The most complex scenario is when the old friends Alice and Bob meet on the Internet and want to establish a secure channel — the key exchange is susceptible to a man-in-the-middle attack. By exchanging questions and answers about common experiences, Alice and Bob can decrease the probability of a successful attack, since it is unlikely that someone could guess all the answers to these questions.

1.1.3 PKI

A PKI is a digital infrastructure that provides security services based on public-key cryptography to an information system. At the core of this infrastructure is the concept of a digital certificate: a data structure binding an entity's name with a public key, digitally signed by a (trusted) party [2].

Centralized PKIs

Typical business-oriented PKIs are centered around a *certification authority*, or CA. The CA is a trusted authority that creates certificates for the users' public keys. The public key of the CA itself is distributed in a secure, out-of-band procedure to the users, so that they are able to verify the signature on the certificates. In large PKI systems, there may be a hierarchy of CAs, each responsible for certifying a subset of the users. Certificates are typically stored in a *certificate repository*, like a phone book where users can retrieve the certificates of others.

Public keys can not be used indefinitely. As cryptography and cryptanalysis advance, and computers become more powerful, key lengths may need to be adjusted. It is also common to limit the amount of data protected by a single key. Therefore, certificates typically expire at a certain point in time, marked on the certificate. After this time the certified key is not accepted for use. However, expired certificates must still be accessible to decrypt data or to verify signatures that were made before the key expired. This *key history* management is a service provided by most PKIs. A *time-stamping service* supplies a trusted, common reference time source and adds signed time stamps to documents whenever necessary.

It is also necessary to include a revocation mechanism in the PKI. Keys may be compromised, or simply not be needed any more, and the CA must be able to revoke certificates for such keys, rendering them unusable even

[2]Numerous certificate formats exist for different PKI systems, e.g. X.509, SPKI/SDSI, PGP etc. These formats define various extensions and attributes as part of the certificate. In this context, we only care about the binding that is made between a user and a key, and the signature on this binding. Therefore our certificates are abstracted to include only these bare necessities.

though they have not yet reached expiration. The most common mechanisms for spreading revocation information are *certificate revocation lists* (CRLs), *certificate revocation trees* (CRTs), and the *online certificate status protocol* (OCSP). CRLs are periodically issued lists of revoked certificates, and have many variants such as redirect CRLs, indirect CRLs, delta CRLs etc. Every time a certificate is used, the most recent CRL is checked to see that it has not been revoked. CRTs are also issued periodically, but are based on the hash tree, a data structure that is more efficient than a CRL. OCSP, on the other hand, is a real-time protocol where users request certificate status information online. A responder server gives signed responses to these requests. Given that the responder has access to fresh revocation information, the latency of this protocol is lower than that of the periodically issued mechanisms, but signing each request slows down performance and enables denial-of-service attacks.

The most important centralized PKI models are based on X.509 [X50900], an ISO/ITU-T standard specifying the formats of general certificates and CRLs. The IETF (Internet Engineering Task Force) working group responsible for X.509 certificates is known as PKIX [PKIX05]. The PKIX work specifies a PKI structure for the Internet, based on X.509 certificates, but including many other parts, e.g. protocols for certificate management, certificate policy framework, time-stamping protocols etc.

The ISO Technical Committee 68 has done work on standardizing a PKI for the financial industry, also based on X.509 certificates.

Decentralized PKIs

A PKI does not have to be based on a single, central CA or a CA hierarchy. An alternative is to let all users act as certificate authorities in a decentralized PKI. In this scenario users issue and disseminate certificates directly to one another, either on-line or through some out-of-band procedure. Decentralized PKI models are mostly used in user clusters based on mutual acquaintances.

One of the most well-known frameworks is OpenPGP [OpenPGP05], an IETF standard based on the older PGP (Pretty Good Privacy) model. OpenPGP certificates bind a public key to a person, identified by a UserID. The UserID is chosen by the keyholder and consists of a common name and an email address. Since email addresses are based on DNS (Domain Name System) which provides globally unique identifiers, the name space is truly global. OpenPGP is a popular framework within the Internet community.

In OpenPGP certificates are issued freely by users, so when Bob looks for Alice's public key he may have a number of certificates for it, each signed by a different user. The problem for Bob is to judge whether or not he can trust these signers. To this end, OpenPGP incorporates a *Web of Trust*, a fault tolerance mechanism to help users make acceptance decisions. Bob has certain friends and acquaintances within the system, whom he trusts to different degrees. If a sufficient number of these known users attest the validity of Alice's key (e.g. three totally trusted users, or six marginally trusted users), Bob will accept Alice's key.

Revocation of an OpenPGP certificate is typically done by its owner, i.e. the holder of the public key pair, or a user whom the owner has designated as a revoker. The revocation is communicated to other users by posting the information on a keyserver [3]. This procedure is called *key revocation*. It is also possible for signers of a certificate to revoke their signatures if they no longer believe in the binding, in a procedure called *signature revocation*.

SPKI/SDSI [EFL+99, CE04] is a decentralized PKI with its roots in the research community. It has a local naming scheme, and supports two types of certificates: one for defining local names and — unlike the other PKI models described here — one for bestowing authorization on a user. With authorization certificates user can delegate authorizations to others, with or without a grant option (a right to delegate further). The authorization must be accepted by the reference monitor for the resource. For Alice to prove that she is authorized access, she must prove that there is a chain of local names from one of the entries in the ACL (access control list) for the resource, to a key that she possesses.

Revocation in SPKI/SDSI is handled by CRLs. Each certificate can only be revoked by a specific key, given in the original certificate. The model only allows one CRL at a time, signed by a given key. If there is a valid CRL signed by the revocation key of certificate c_i, that does not include c_i, then c_i is concluded to be valid.

SPKI/SDSI has not reached widespread usage, but the model has generated a fair amount of research [CE96, TA98, JRH00, CEE+01, HvdM03].

Note that a decentralized PKI can have either a local or a global name space. Decentralization in this context only refers to the fact that all users can issue certificates.

[3]One of the reasons for a revocation is that the private key could have been lost, keeping the key owner from decrypting messages to them, and what is worse: keeping them from signing a revocation certificate. To preclude this situation, revocation certificates should be created at key generation time, and stored offline until needed.

1.2 Motivation

Consider figure 1.2, a simple graph that is a first introduction to our formalism (more attributes will later be added to the graph elements, but they are not necessary here). The circular nodes represent the entities A, B, C, and D, and the boxes on the edges between them represent a digital certificate for B's public key $B.K$, signed with A's private key $A.k$. The certificate is passed from A to B, from B to C, and then on to D.

The reason that B can pass the certificate on to C is that they have first received it from A. Since A is the originator, every copy of the certificate must be connected to A by a path of certificates. If A were to take back the information from B — because it is no longer valid for some reason — the information to C should also be removed. Following this, the certificate between C and D must also be removed. This is a basic example of revocation.

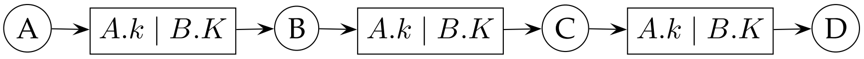


Figure 1.2. Entities spreading information

Now consider figure 1.3. In this graph, C receives a copy of the certificate directly from A, in addition to the one they get from B. If A were to revoke B's copy of the certificate, and hence the edge between B and C was removed, C still has the information directly from A and can therefore still tell D about it. In this case, the edge from C to D should not be removed.

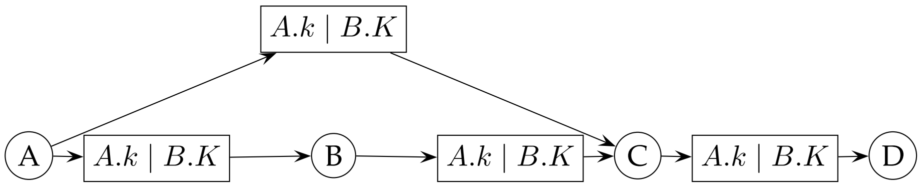


Figure 1.3. Receiving information from several sources

The Merriam-Webster dictionary explains the act of revoking as "to annul by recalling or taking back". Thus, revocation of a certificate could be the act of a user who recalls a certificate previously passed to another user. Somehow, the revocation must cascade in the system to make sure that no information is derived from obsolete certificates.

This description of revocation seems simple enough at first glance. However, even in such a specific environment as a PKI — where all the information passed consists of certificates, each on the same form — one has to be very careful when defining *what* is being revoked. Is it the key itself? Is it the binding between the user and the key? Similarly, there can be a number of reasons why a revocation should take place. The key may have compromised, or the owner may simply not need the key any longer.

The simplest way to revoke a certificate is to remove it from the system, but that is not the only way to annul the information it represents.

Expiration, or time-out, of a public key is one way to remove a valid key from the system. Thus, we regard it as a form of revocation.

A stronger way to revoke a certificate is to issue its inverse; if there was previously a certificate binding B and their public key $B.K$, the inverse would be a certificate stating that $B.K$ is *not* B's public key. Note that this annulment will be time-persistent in the sense that any subsequent certificates on the same form as the first (positive) one, will have to deal with the presence of the negative certificate.

Entities can use the information in a certificate to deduce new information. For example, if Alice receives a certificate that Bob has signed for Carol's key $C.K$, and Alice knows Bob's public key, she can verify the signature on the certificate and deduce knowledge about $C.K$. When a certificate is revoked, the information obtained using the revoked key should be removed as well. The extent of the subsequent removals depends on the reason for the revocation. If the key has expired (but was valid at one time), information derived from the previous knowledge of the key may still be valid, but no additional information should be derived using the obsolete key. If the key has been compromised (and therefore may not have been valid in the past) then other certificates derived using this knowledge should be recursively revoked.

We consider any kind of annulment of information — whether by removal, expiration or by issuing the inverse — to be a form of revocation, and investigate the results of all these actions.

Our aim is to understand the meaning of revocation in the context of a decentralized public-key infrastructure — not to find an efficient implementation for it, but to investigate the implications of a revocation and how these implications depend on the reasons for the revocation. While revocation is our main focus and concern, certificate distribution must also be modelled. The reason for this is twofold: in part to show the structure of chains that revocation must act upon, in part to show what actions entities are allowed to perform on revoked certificates of different types.

1.2.1 Cycles

Unlike hierarchical models with a CA that distributes every certificate, in a decentralized PKI care must be taken to avoid cycles in certification paths. Consider figure 1.4, a graph describing the spreading of a certificate for $B.K$, signed with $A.k$. The edge from A to A marked with a double box represents their outside public-key knowledge of $B.K$. In other words, A has established confidence in $B.K$ through some secure out-of-band procedure. The other edge from A to A represents A's knowledge of their own keypair (k, K). A uses this private key and signs a certificate for $B.K$, which is distributed to C, who in turn quotes it to other entities. A's public-key knowledge must be in place before the certificate can be created or quoted, and it can be viewed as the root of the paths for quotations of this particular certificate.

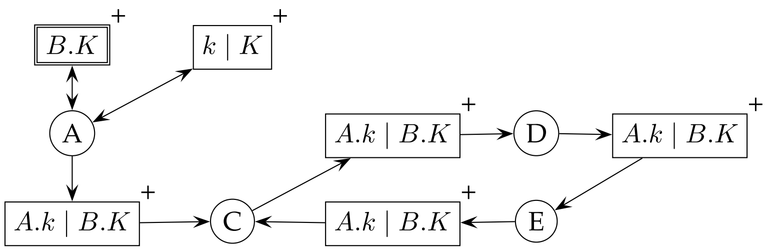


Figure 1.4. A cycle example

From C's point of view, there are two incoming edges with the certificate, and one outgoing edge with a quotation of it. However, from a global point of view, it is only the edge from A that connects C to A's outside knowledge, which supports all the other certificates. Now assume that A removes the edge between themselves and C. C's link to A's outside knowledge has been severed, but C is unaware of this — they still have an incoming edge from E, and as far as C can tell, this supplies support for their outgoing edge. Globally, we can see that there is a cycle involving C, D and E, but in the step-by-step procedure of a revocation only one node at a time is considered. We need a way to capture these types of patterns and deal with cycles when certificates are revoked.

Paths also form through deductions, as shown in figure 1.5. In the figure, an entity A receives a certificate signed by B for $C.K$. Since A has public-key knowledge of $B.K$, they can verify the signature and deduce public-key knowledge of $C.K$. This knowledge depends on B's certificate for $C.K$, so that if A loses that support, the deduction should be removed.

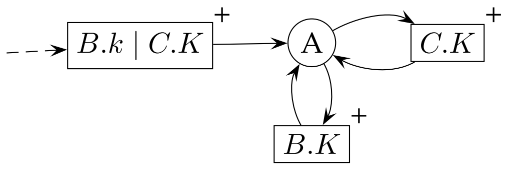


Figure 1.5. Path forming through deduction

If A has spread their knowledge to others by signing a certificate for $C.K$ and a cycle has formed, this must also be detected.

1.3 Our Approach

Many researchers use graphs to illustrate and concretize their ideas. We consider graphs themselves to be a powerful tool for modelling and reasoning about systems, and we have chosen to take advantage of their expressive and intuitive properties. Our formalism of choice is a graph and graph transformation rules. The information state of an abstract PKI is captured in a graph which includes all the entities, and the certificates they have passed to each other. The graph transformation rules define allowed adjustments — additions and revocations — to the knowledge and information, as well as deductions adding new knowledge.

The system we have modelled is not a translation of any existing framework or paradigm (such as X.509 or PGP). Instead, the purpose of the model is to define a decentralized system for certificate distribution and revocation under the given assumptions that users act with *local knowledge only*. There is no central authority with a complete overview, nor is it possible for any entity to take global actions. With the model in place, we investigate what revocation means in this context, and how the assumption of localness has affected the effect of the revocation mechanisms.

No specific assumptions are made on the way distribution or revocation are implemented. In particular, we do not deal with CRLs, which are a specific implementation chosen to represent specific information. Our model is not affected by alternative choices on the distribution of the information about revoked certificates (e.g. broadcasting).

We assume that there is a secure out-of-band method for entities to establish confidence in keys. Keys may be shared via some physical channel, e.g. in a letter or via a phone call, or they may be distributed electronically but verified offline, e.g. by comparing the hash value of the key to the so-called fingerprint of the key, which may be distributed out-of-band.

Our view of identity is that an entity is a collection of public keys. For simplicity we assume a global name space, i.e., a user is known by the same name to every other user.

To handle the cycle problem one can either prohibit cycles to form or handle them at revocation time, making sure that cycles are not considered as support. As we want to allow "free speech" in our system — entities should be able to spread information freely — we have opted for the latter approach. Our solution is to include support sets below every certificate. A support set is a representation of the acyclic paths that connect that certificate to an outside knowledge. This makes it possible to see when a certificate is disconnected from all its supporting paths.

1.4 Outline

In this chapter, we have already presented the background and motivation for this research. The following chapter will present related work. In chapter 3 we will give an introduction to the theory of graphs and graph transformations. This material is largely an overview based on general graph theory, but the definition of the graph morphism and the matching condition have been adapted to suit our purposes. Chapter 4 presents our terminology and gives some definitions, notably the definition of a valid state and the localness assumptions. The C-graph model with graph transformation rules for modelling the distribution and revocation of public-key knowledge and certificates is presented in chapter 5. This chapter constitutes the main part of the work. We give flowcharts that describe how the revocations propagate through the system in chapter 6, where we also analyze the soundness and completeness of the rules with respect to our valid state definitions. Here, the reader can also find a demonstration of parallel and sequential independence within and between the rule layers, respectively. Chapter 7 decribes how some aspects of the model evolved over time, as well as gives suggestions for extending the model. Finally, a discussion and conclusions are given in chapters 8 and 9.

Chapter 2

Related Work

In this chapter we will present previous work that is related to our research. The work has been divided into three categories: first we present other formalisms for modelling and reasoning about public-key certificates; next some work that has been done on cycle detection; and finally related papers on the topic of revocation.

2.1 Formalisms

Numerous models for the reasoning about public-key certificates have been proposed. Many of these researchers use graphs to visualize their ideas, and make them easier to grasp for the reader. When it comes to the formal treatment of rules and reasoning, however, most previous work in this area has used other formalisms, based on logic, calculus or language.

2.1.1 Logic-Based Formalisms

Maurer [UM96] was one of the first to model a PKI using both keys and trust. Alice needs to know Bob's public key, as well as to trust him, in order to believe the statements that he makes. Every statement made by an entity in the system is about keys or trust. Trust is given in levels; a higher level of trust in a user implies the possibility of longer chains of derived statements starting from that user. In the second part of the paper Maurer refines the concept of trust to a probabilistic model, where users can state a trust confidence parameter between 0 and 1 in other users. In Maurer's model, each user's view (including all belief and trust the user has, and all recommendations made to them) is modelled separately from the others. It is therefore difficult to get a global view of the system, and to maintain dependencies between different users' statements.

Stubblebine and Wright [SGS95, SW96] describe a logic for analyzing cryptographic protocols that supports the specification of freshness constraints on protocols. Assuming that information about revoked keys cannot be immediately distributed to all parties of a system, they instead focus on policies for decisions based on information that may be revoked. Simple examples of such policies are *believe if recent* and *believe until revoked*. The authors' model allows reasoning about revocation of keys, jurisdictions, and generally, of arbitrary beliefs. Users are assumed to communicate honestly.

Li et al. [LFG99, NL00] present a logic-based knowledge representation for distributed authorization and delegation. The logic allows more general statements than simple beliefs about public keys and trust, and it lets users reason about other users' beliefs. The authors concern themselves with the problem of non-monotonicity and use overriding policies to determine which statements take precedence. It seems difficult to completely remove statements and their consequences from the system, however, something that might be desirable from a revocation perspective.

Liu et al. [LOC01] use a typed modal logic to specify and reason about trust in PKIs. Trust and belief in public keys are both included in the formalism. A certification relation and a trust relation are used to specify which entities are allowed to certify other entities' keys, and which users they trust, respectively. The former is static, while the latter may change dynamically. In order to accept a statement, a user must find a path to that statement starting with a trusted certificate. If no such path can be found, the statement is not accepted. Revocation is enforced by an overriding policy — users that wish to revoke a certificate add it to their CRL, which overrides previous information.

Halpern and van der Meyden [HvdM03] make a logical reconstruction of SPKI/SDSI, including revocation and expiration of certificates. CRLs are modelled as a signed set of certificates, and expiration is achieved through validity intervals. The model is monotonic due to the fact that in SPKI/SDSI, a certificate is ignored unless it can be shown not to be revoked. Halpern and van der Meyden argue that non-monotonic logic is not required "if one takes the SPKI perspective that revocation is not a change of mind but a revalidation". The proof for the validity of a certificate is the fact that it is not present in any valid CRL.

2.1.2 Calculus-Based Formalisms

Kohlas and Maurer [KM99] propose a calculus for deriving conclusions from a given user's view, which consists of evidence and inference rules that are valid in that user's world. Statements can be beliefs or recommendations about public keys or trust, commitments to statements and transfer of rights (delegation). There are no negative statements or other possibilities for revocation.

2.1.3 Language-Based Formalisms

Gunter and Jim [GJ00] define the programming language QCM, used to define a general PKI with support for revocation and delegation. The authors note that "part of the confusion regarding revocation and PKIs stems from treating revocation data specially [. . .] data used for revocation should be treated *dually* to other sorts of information"; in their model, revocation is handled by negative statements and an overriding policy. QCM does not require a specific distribution mechanism, but separates the implementation from the specification of revocation. The authors remark that "a PKI must unambiguously specify how revocation should be interpreted".

2.1.4 Graph-Based Formalisms

The work of Capkun et al. [CBH03] is the most closely related to ours. The authors describe a "self-organizing public-key management system", that lets users of a mobile network "create, store, distribute and revoke their public keys without the help of any trusted authority". Public keys and certificates are described as a directed graph G, where the nodes represent public keys and the edges represent certificates: an edge from node K_u to node K_v represents a certificate signed with the public key of u, binding K_v to an entity. Upon creation of a certificate, the signer and the subject both know about it and later spread information about the certificate to other entities.

Entities are not represented in G, but store their own knowledge in two graphs each: the updated and the nonupdated certificate repositories (G_u and G_u^N, respectively). These repositories are partial views of the system graph G, so that G_u (of user u) has an edge between nodes K_v and K_w if u knows about v's certificate for K_w. The non-updated repository is added to periodically when users exchange subgraphs with their physical neighbors, but ones already in the graph are not updated at this time. Thus, it may contain recently expired certificates. The updated certificate repository contains only valid certificates — users register with certain certificates' issuers to be notified when these certificates are revoked or updated.

To verify the key of another user v, u creates the union of G_u and G_v (first requesting G_v from v), and then attempts to find a path from K_u to K_v. Failing this, u creates the union of G_u and G_u^N, and again attempts to find a path. If a path is found, the certificates from G_u^N that were used are checked for validity.

Revocation takes place if a user believes that a certificate they issued has lost its validity, or if they believe that their own public key has been compromised. To revoke a certificate, a user can choose between *explicit* and *implicit revocation*. With explicit revocation, the user sends a revocation statement to other users who have requested to be informed about it. Implicit revocation takes place automatically when the validity period of a certificate is over, unless the certificate is renewed. To revoke their own public key, the user notifies the users who have signed certificates for that key, and these users then proceed with explicit revocation of those certificates.

Capkun et al. note some attacks where malicious users issue false certificates, perhaps to impersonate other users.

The results of the paper consist of simulations of the algorithms described, along with performance analyses. The authors also analyze the problems of minimizing the sizes of repositories and key usage.

2.1.5 Other Models

The models mentioned in the preceding sections have all been used to model certificate distribution and/or revocation, much in the same way that we need for our purposes. There has also been some work in this field that uses graphs in a formal way, but that is further away from our basic problem formulation:

Wright et al. [WLM00, WLM01] present a decentralized model. They define *depender graphs* — rooted, directed, acyclic graphs where every node except the root and its dependants have k parents. The nodes are users in a PKI, and there is an edge from A to B if B is a depender of A, i.e. if A has agreed to forward revocation information to B about a specific certificate. Each certificate has its own graph. Graph properties are used to analyze the system, and the depender graph is shown to have k-redundancy — the system guarantees revocation notification to all on-line participants even when $k-1$ participants are unavailable. The model is localized, i.e. no global view of the graph is maintained.

Buldas et al. [BLL00, BLL02] introduce *authenticated search trees* to model undeniable attesters — this is a primitive that is used for long-term certificate management supporting key authenticity attestation and non-repudiation. Tree properties are used to analyze the complexity of the model.

2.2 Cycles

A few papers have been published, where cycles and paths are mentioned in relation to public-key certificates and revocation.

Aura [TA98] defines *delegation networks*, which are directed bipartite graphs used to pass authorizations between users. Although the authorizations are transferred in certificates, the usage of keys and signatures has been abstracted away. Graph searching algorithms are used to find support for authorizations.

Aura explicitly allows cycles in a delegation network, i.e. a key can delegate authorizations to itself, either directly or indirectly. The reason to allow cycles is to avoid complexity in the definitions. Revocation is not discussed in any detail.

PKIX/X.509 [HFPS02, CDH$^+$05] includes a procedure for *certification path validation*, a process which establishes a path between the certificate at hand and a certificate signed by the *trust anchor*, e.g. the top CA. The PKI is represented as a graph with entities as nodes and certificates as edges — note that an edge from node A to node B represents a certificate signed by A for B's public key, not information passed between them. To validate a path, the process must ascertain that the first certificate was issued by the trust anchor, that the subject of a certificate in the path is the issuer of the subsequent one, and that all certificates in the path are valid. There may also be policies in place that specify which possible paths are accepted and which are not. Housley et al. note that "the trusted anchor information is trusted because it was delivered to the path processing procedure by some trustworthy out-of-band procedure".

The X.509 specification [X50900] does not allow certificates to repeat in a certification path. Cooper et al. [CDH$^+$05] discuss loops (cycles) forming in paths, and note that in bridged PKI environments, different certificates for the same entity may be involved in a loop. Although this would be compliant with the X.509 specification, it is an undesirable situation. The authors therefore recommend disallowing pairs of public keys and subject names from being repeated in a path.

2.3 Revocations

The notion of revocation is hard to grasp, and various meanings can be given to the concept. Some previous work has examined different types of revocation, where the desired results typically depend on the reason for the revocation.

Cooper [DC98] divides revocation reasons into benign and malicious types, and notes that different revocation practices are needed for the two types. Particularly, when on-line renewal of certificates is allowed in a system, and a certificate is revoked for a malicious reason (e.g. because of key compromise), there is a risk of attackers impersonating the real key owner. All certificates created through on-line renewal of the revoked certificate must also be revoked to avoid the attack.

Fox and LaMacchia [FL98] note that "revocation of public key certificates is controversial in every aspect: methodology, mechanics, and even meaning". They discuss different reasons why a public key certificate might need to be revoked. The meaning of a revocation could be to no longer trust the key because it has been compromised, to no longer trust the binding between key and subject because it is no longer valid, or to no longer trust the relationship between the issuer and the certificate because the issuer no longer vouches for the binding. The authors note that different revocation mechanisms are necessary for the different reasons.

Rivest [RLR98] suggests that CRLs do not constitute a good revocation mechanism, and proposes instead that the signer using a key should supply the necessary evidence of its validity, instead of the other way around. Short-term certificates are proposed as good evidence for recent validity. McDaniel and Rubin [MR99] give a response to Rivest, where they note that CRLs are useful in tightly coupled environments, and propose a mechanism for revocation on demand, where CRLs are issued and distributed at predetermined intervals.

Khurana and Gligor [KG00] discuss revocation of access privileges that have been distributed as attribute certificates within a PKI. Privileges can be shared via delegation certificates, thus forming delegation chains. When several types of certificates are used in a system (e.g. attribute, identity and delegation certificates), the dependencies between the types must be considered at revocation. The authors propose *selective revocation*, where attribute certificates of users whose identity certificate has been revoked are selectively revoked as well. They also note that *transitive revocation* is necessary to revoke delegation chains.

Li et al. [LF01] argue that revocation is complex and confusing due to three reasons: revocation makes certification non-monotonic with respect to time; the user interface and the internal mechanisms of a PKI are often confused; revocation is viewed as a way of providing security, instead of a method of controlling risks. To make revocation less confusing, the authors give seven recommendations for how a PKI should handle and present revocation information.

The PKIX/X.509 certificate and CRL specification [HFPS02] defines nine reason codes for revocation of a public-key certificate, but does not suggest different revocation practices for different codes — the only revocation method is when the CA adds an entry to the CRL. The reason codes are defined as non-critical extensions:

(1) keyCompromise
(2) cACompromise
(3) affiliationChanged
(4) superseded
(5) cessationOfOperation
(6) certificateHold
(7) removeFromCRL
(8) privilegeWithdrawn
(9) aACompromise

The OpenPGP specification [CDFT05] also defines reason codes:

(1) No reason specified
(2) Key is superseded
(3) Key material has been compromised
(4) Key is retired and no longer used
(5) User ID information is no longer valid

The last of these items is used for signature revocation, i.e. when the signer of a certificate revokes their signature. The others are used for key revocation — when the owner of a key revokes it.

The specification notes that revocations should be interpreted differently, according to the reason code given:

> If a key has been revoked because of a compromise, all signatures created by that key are suspect. However, if it was merely superseded or retired, old signatures are still valid. If the revoked signature is the self-signature for certifying a User ID, a revocation denotes that that user name is no longer in use. [...]
>
> Note that any signature may be revoked, including a certification on some other person's key. There are many good reasons for revoking a certification signature, such as the case where the keyholder leaves the employ of a business with an email address. A revoked certification is no longer a part of validity calculations. [CDFT05]

Hagström et al. [HJPPW01] define and classify different types of revocation schemes for an ownership-based access control system using the dimensions *resilience*, *propagation* and *dominance* — each dimension is binary, so the combination of all possibilities results in eight types. Permissions can be delegated with or without a grant option, thus forming delegation chains. Revocation is done either by removal or by issuing negative permissions; both propagate in the delegation chains but in different ways depending on the chosen revocation scheme.

Resilience describes the difference between revocation via removal and revocation via negative permissions. The *delete* action is local in time — no trace remains of the previous revocation, thus it is not resilient. A *negative* permission, on the other hand, remains in the system, and will overrule new positive permissions given even after the revocation had occurred.

Propagation describes how a revocation spreads via delegation chains. A *local* revocation is intended only for the direct recipient of a permission, whereas a *global* revocation reaches all other users in turn authorized by the direct recipient.

Dominance describes how a revocation deals with conflicts that arise when the subject losing a permission through revocation still has permissions from other grantors. If the other grantors have received their permissions from the revoker, they can be dominated in a *strong* revocation. In a *weak* revocation, only permissions that come directly from the revoker are removed.

Chapter 3

Graph Concepts

In this chapter we review basic concepts of graphs and graph transformations. We use the single-pushout (SPO)[1] approach to graph transformations; details are given by Löwe and by Ehrig et al. [ML93, EHK^{+}97]. Rudolf and Taentzer [RT99] offer a more accessible account of the theory. An alternative to SPO is the classical double-pushout (DPO) approach [CMR^{+}96], but is has been shown that the SPO approach is a generalization of DPO and that important results from DPO research can be extended into SPO frameworks [ML93].

3.1 Graphs and Graph Morphisms

A graph describes a relation where pairs of vertices (nodes) are connected by directed edges — each edge has a source and a target node. In an attributed graph, nodes and edges have attributes from the predefined sets *V-ATT* and *E-ATT*, where each attribute is a tuple of values from fixed alphabets. More formally:

Definition 1 (Attributed Graph)**.** *An* ***attributed graph*** *G over the attribute sets $(V\text{-}ATT, E\text{-}ATT)$ is a six-tuple $G = (V, E, s, t, v\text{-}att, e\text{-}att)$ where V and E are finite sets of vertices and of edges and $s, t : E \rightarrow V$ assign source and target nodes, respectively, to each edge. The functions $v\text{-}att : V \rightarrow V\text{-}ATT$ and $e\text{-}att : E \rightarrow E\text{-}ATT$ assign attributes to vertices and edges, respectively.*

[1]The name single-pushout comes from category theory, which is used for the analysis and formal treatment of graph transformations (figure 3.1 is in fact a pushout diagram in the category of graphs and graph morphisms). We will not delve into category theory, but it is useful to know that it is the foundation of graph transformation theory.

A *subgraph* S of G (denoted $S \subseteq G$) is a graph that consists of subsets of the vertices and edges of G, connected and attributed identically to the corresponding elements in G.

As is common, we denote the domain of a function f — i.e. the elements for which f is defined — with $dom(f)$.

A *morphism* between two graphs over the same set of attributes consists of four functions that preserve the structure of the graphs:

Definition 2 (Graph Morphism)**.** *A graph morphism $f : G_1 \to G_2$ between the attributed graphs $G_1 = (V_1, E_1, s_1, t_1, v\text{-}att_1, e\text{-}att_1)$ and $G_2 = (V_2, E_2, s_2, t_2, v\text{-}att_2, e\text{-}att_2)$, both over the attribute sets $(V\text{-}ATT, E\text{-}ATT)$, consists of four functions:*

$$f = \begin{cases} f_V : V_1 \to V_2 \\ f_E : E_1 \to E_2 \\ f_{v\text{-}att} : V\text{-}ATT \to V\text{-}ATT \\ f_{e\text{-}att} : E\text{-}ATT \to E\text{-}ATT \qquad \text{such that:} \end{cases}$$

$$\text{(1)} \ \forall\, e \in dom(f_E) : \quad f_V(s_1(e)) = s_2(f_E(e))$$

$$\text{(2)} \ \forall\, e \in dom(f_E) : \quad f_V(t_1(e)) = t_2(f_E(e))$$

The two characteristics of f_V and f_E imply that a morphism must be compatible with the structure of the graphs G_1 and G_2. In other words, if f_E maps the edge e to the edge e', then f_V must be defined for the source and target nodes of e, and map them into the source and target nodes of e', respectively.

Definition 2 is less complex than the corresponding definitions given in related work, e.g. by Ehrig et al. [EHK+97, EPT04]. The attributes needed for our purposes are simpler than the general case considered by other researchers — we only need constants and variables as attribute values, not evaluation of terms. Therefore, we chose to do without signatures, categories and algebras.

A *partial graph morphism* $g : G_1 \rightharpoonup G_2$ is a graph morphism from some subgraph of G_1 to G_2. The subgraph is the domain of g, $dom(g)$. When a graph morphism $g : G_1 \to G_2$ is *total*, $dom(g) = G_1$.

3.2 Graph Transformation Rules

Graph transformation rules (also called productions) can be used to construct and modify graphs. Put simply, a rule consists of two graphs, describing the state of a *host graph* before and after the desired operation. The objects in the graphs are abstract variables that are instantiated when the rule is applied to a concrete graph. More formally:

Definition 3 (Graph Transformation Rule). *A **graph transformation rule** is an injective partial graph morphism $r : L \rightharpoonup R$, where L and R are graphs called the **left-hand side** and the **right-hand side** of the rule.*

L describes the state of a graph before the rule is applied, and R describes the desired state afterwards. Only objects and attributes that are relevant to the rule are included in L. Elements of L that are not present in R are deleted by r; elements that are present in both L and R are kept. Since r may be undefined for some elements of L (i.e. those that are deleted by the rule), r is a partial morphism. It is injective because we require each object in L to have its own image in R.

3.2.1 Matches and Derivations

$$\begin{tikzcd} L \arrow[r, "r"] \arrow[d, "m"'] & R \arrow[d, "m*"] \\ G \arrow[r, "r*"'] & H \end{tikzcd}$$

Figure 3.1. A rule $r : L \rightharpoonup R$, applied to the host graph G, resulting in H

The application of a rule is called a *derivation*. Figure 3.1 describes the application of $r : L \rightharpoonup R$ in a host graph G. The application requires an occurrence of L in G — a total morphism $m : L \rightarrow G$, called *match morphism* ($m(L) \subseteq G$). This match morphism is total because all conditions imposed by the rule must be satisfied, i.e. all elements of L must have an image in G. The mapping $m^* : R \rightarrow H$ is a related morphism called the *co-match* of the derivation. H, the *derived graph*, is obtained by replacing the occurrence of L in G by R through the *co-production* $r^* : G \rightarrow H$.

The actual transformation of the host graph G is performed in two steps: the match of L in G ($m(L)$) is found, and the elements of $m(L \setminus dom(r))$ (the elements of the match for which r is not defined) are removed from G. Next, the elements of $R \setminus r(L)$ (the elements that have no preimage under r) are added to the host graph, resulting in H. The elements that are preserved by r form the *application context*, which is used to connect the new elements to the host graph. Edges that are left dangling after the transformation (without either a source or a target node, or both) are deleted.

Graph transformation rules can also manipulate the attributes of edges and nodes, by using expressions for the right-hand side attributes. These expressions are evaluated with respect to the variable instantiation of the match morphism.

The match morphism m need not be injective; different objects in L may be mapped onto the same object in G. Conflicts may arise when m is non-injective, for example when two objects in L are mapped to the same object in G, and one of these objects is deleted by the rule r while the other is preserved. In these cases deletion takes precedence. For this reason, the co-match $m^* : R \to H$ is a partial morphism, since elements of R that are deleted because of conflict do not have an image in H.

When there are no matches of L into G, r is not applicable. There may also be several possible matches of L into G — in this case, one of the matches must be chosen, either at random or interactively. In our system, certain rules are intended to be called by a user or an administrator specifying a single match (interactive rules), and others are intended to be applied automatically (deductive rules). The deductive rules are matched at random into the host graph.

The Matching Condition

We need a *matching condition* on the application of graph transformation rules to make sure that rules and matches work together as intended:

Condition 1 (Matching Condition). *Given the pair of a production and a match $(r : L \to R, m : L \to G)$ of a derivation, the following must hold:*

(1) $\forall\, a_v \in \{v\text{-}att_L(v) \mid v \in dom(r_V)\}$:

$$\begin{aligned} r_{v\text{-}att}(a_v) &= a_v, \quad \textit{or} \\ m_{v\text{-}att}(a_v) &= a_v \end{aligned}$$

(2) $\forall\, a_e \in \{e\text{-}att_L(e) \mid e \in dom(r_E)\}$:

$$\begin{aligned} r_{e\text{-}att}(a_e) &= a_e, \quad \textit{or} \\ m_{e\text{-}att}(a_e) &= a_e \end{aligned}$$

The requirements describe the same condition for nodes and edges, respectively: if an item (node or edge) is not removed by a rule $r : L \rightarrow R$, then in a derivation with the match morphism $m : L \rightarrow G$, each attribute value of that item must be preserved by either r or m (or both).

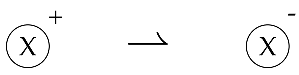


Figure 3.2. A rule $r : L \rightarrow R$ that preserves the name and changes the state of a node

To see why, consider nodes that have two attributes: a *name* (shown inside the node) and a *state* (shown to the upper right of the node). Consider a rule $r : L \rightarrow R$ that changes the value of the state attribute of a single such node n from $+$ to $-$, but keeps the name attribute unchanged (as shown in figure 3.2). The value of the name attribute is unimportant in the rule, so it is given in the form of a variable. To keep the name attribute intact, the name of n is the same variable in L and R — i.e. the attribute is preserved by r. Since it is given as a variable in L, the match morphism can change the value of the name attribute — from an unspecified value in L to a specific value in G. In this case, r preserves the attribute but m changes it.

Now consider the state attribute. In L, the state of n is $+$, and in R, the state is $-$. In order for the rule to work as intended, m must preserve the value of the attribute — i.e., the node that n is matched to via the matching morphism must have a state that has the value $+$. Otherwise, the rule could be applied to a node where the state has another value, which is not what it is intended for.

In other words: if the value of an attribute is given as a variable in L, r must preserve the value; if the value of an attribute is given as a constant in L, m must preserve the value.

Example

To illustrate how a matching is done, we will give an example of a rule and its matching into a host graph.

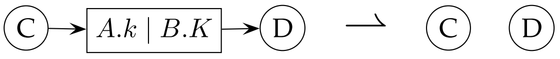


Figure 3.3. Example rule — removing a certificate

In figure 3.3, the left-hand side contains the entities C and D (represented by circular nodes), and a certificate where A vouches for $B.K$, passed from C to D (represented by a box on the edge between them). When this rule is applied to a specific graph, C, D, $A.k$ and $B.K$ must all be instantiated to nodes and edges present in that graph, thus matching the left-hand side. The effect of the rule is to remove the edge (cf rule 19), and this rule is to be called interactively by a user, the entity which instantiates C in the host graph.

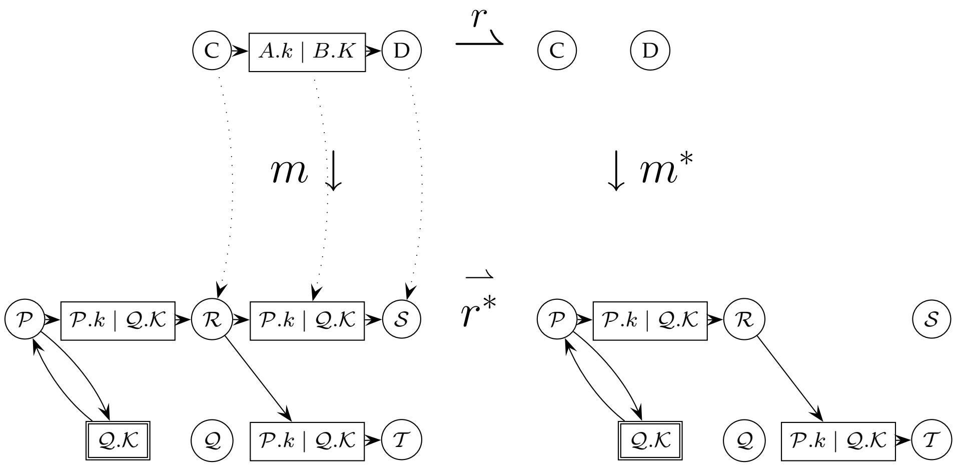


Figure 3.4. Matching a rule in a host graph

Figure 3.4 illustrates the matching and the effect of this rule when it is applied in a host graph (on the lower left). The matching is shown with dotted arcs. In this case, the user $\mathcal{R}$ calls the rule, and specifies that $A.k$, $B.K$, and D should be matched into $\mathcal{P}.k$, $\mathcal{Q}.\mathcal{K}$, and $\mathcal{S}$, respectively. C is automatically matched into $\mathcal{R}$ because they applied the rule. The effect of the rule in the host graph is to remove the specified edge between $\mathcal{R}$ and $\mathcal{S}$, just as the effect of r is to remove the specified edge between C and D.

Note that no other elements in the host graph are affected, and that there are two other places in the host graph where this rule could also have been matched (remember that m need not be injective, so C and A could both be mapped into $\mathcal{P}$ if desired).

3.2.2 Negative Application Conditions

The left-hand side of the rule in figure 3.3 specifies necessary conditions for the rule to be applicable — in other words, the left-hand side is an application condition. To make rules more expressive, they can also be equipped with *negative application conditions* (NACs) which specify elements that must *not* be present for the rule to apply.

A NAC for a rule $r : L \rightharpoonup R$ is a set of constraints. These constraints are total injective morphisms $c_i : L \rightarrow N_i$. N_i represents a forbidden structure by identifying a subgraph that must not be present in G for r to be applicable. Matches for r that include the elements of $L - N_i$ are not valid.

Definition 4 (Constraint Satisfaction). *A match $m : L \rightarrow G$ for a rule $r : L \rightarrow R$ **satisfies** the constraint $c_i : L \rightarrow N_i$ if there is no total morphism $d_i : N_i \rightarrow G$ such that $d_i \circ c_i = m$.*

In other words, if the matching m cannot be extended to include N_i, the constraint c_i is satisfied and the matching is valid. We require d_i to be injective in order to prevent elements of L to be mapped into the same element as one from N_i when they are not explicitly marked with the same variable name.

When a NAC consists of several constraints, all these constraints must be satisfied — i.e. none of the forbidden structures must be present — for the NAC to be satisfied.

In some frameworks, each constraint is drawn as a separate graph and presented together with the corresponding rule r [AGG05]. In our diagrams, we include the constraints in the left-hand side of a rule. We denote the rule $r : L \rightharpoonup R$ with NAC $c : L \rightarrow N$ by representing the left-hand side with N, with its L-part drawn solid, and the $N - L$-part drawn dotted. In other words, the parts that must not be present for the rule to apply are drawn dotted. In the case when a NAC consists of several constraints, each constraint is enclosed with a dotted circle, for clarity. Within a constraint that consists of several elements, all elements must be present for the constraint to be violated, i.e., if one of the elements within the circle is missing, then the constraint is satisfied.

When a node or an edge of a NAC is marked with an attribute, it indicates a specific attribute value which is forbidden by the NAC. The constraint is satisfied unless the element can be matched with that particular attribute value. In other words, we can prevent a specific matching to take place. When the attribute of a node or an edge bears no importance in a constraint, the attribute is not included.

We extend the definition of a graph transformation rule to include NACs.

Example

Figure 3.5 shows a rule with a single-constraint NAC (cf rule 7). The interpretation of the left-hand side is that C, D, E, $A.k$ and $B.K$ must be matched into the host graph, and that there must not be a match for a certificate $(A.k, B.K)$ being passed from D to E. Note that E may receive other certificates from D, and that D may spread that certificate to other entities; the constraint only forbids the specific edge with source node D, target node E and attributes $(A.k, B.K)$. If these conditions are satisfied, the rule can be applied and the certificate will be added between D and E. The reason for a NAC such as this one is to prevent duplicates in the graph.

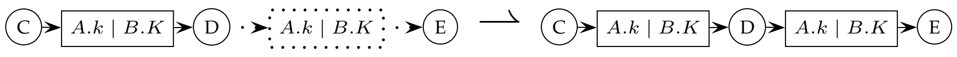


Figure 3.5. A Negative Application Condition

For an example of a multi-constraint NAC, see rule 14, where each constraint is enclosed with a dotted circle. All three constraints of the rule must be satisfied for the rule to apply.

3.2.3 Rule Expressions

Rule expressions are a high-level construct, used to control the application of graph transformation rules. For our purposes we only need expressions of the form `asLongAsPossible` r `end`. This expression applies the rule r until there are no more ways to match the left side of r into the host graph. Bottoni et al. [BKPPT05] give more details on rule expressions.

3.2.4 Properties of Graph Transformations

The underlying theory of the SPO approach ensures some desirable properties of graph transformations [RT99]:

(1) Completeness — all effects specified in the rule are actually performed in the concrete derivation.

(2) Minimality — nothing more than what is specified in the rule is done (with the well-defined exception of the implicit removal of dangling arcs and conflicting objects).

(3) Localness — only the fraction of the host graph covered by the match (including potentially dangling arcs) is affected by the transformation.

3.3 Conflicting Rules

In a system with many graph transformation rules, it is possible that rules conflict with each other in unexpected ways. For example, applying rule A followed by rule B to a graph G might give a different result compared to applying first rule B, then rule A. This may happen in the case where rule A changes an attribute that appears in the left-hand side of rule B. Another possibility is that rule A removes an application condition for rule B, with the result that rule B is applicable before, but not after, rule A.

To help prevent and analyze potential conflicts we introduce the notions of *layers* and of *independence*. The concept of independence between rules can be considered from two different points of view: parallel and sequential independence.

3.3.1 Rule Layers

To avoid rule conflicts and ensure the predictability of a model, a set of rules can be ordered in *layers* $L_1, L_2 \ldots L_n$, which provide a control flow mechanism (see figure 3.6 for an example layering). The layers keep the rules separated — instead of matching the rules at random all at once, only rules in one layer at a time are matched. Within the layers, rules are matched at random. The layers are applied in order and as long as possible: first apply rules of layer L_1 as long as possible, then rules of layer L_2 etc. It is necessary to prove that the rules within each layer may be applied in any order with a deterministic outcome, and that the rules of subsequent layers do not affect the applicability of previous layers.

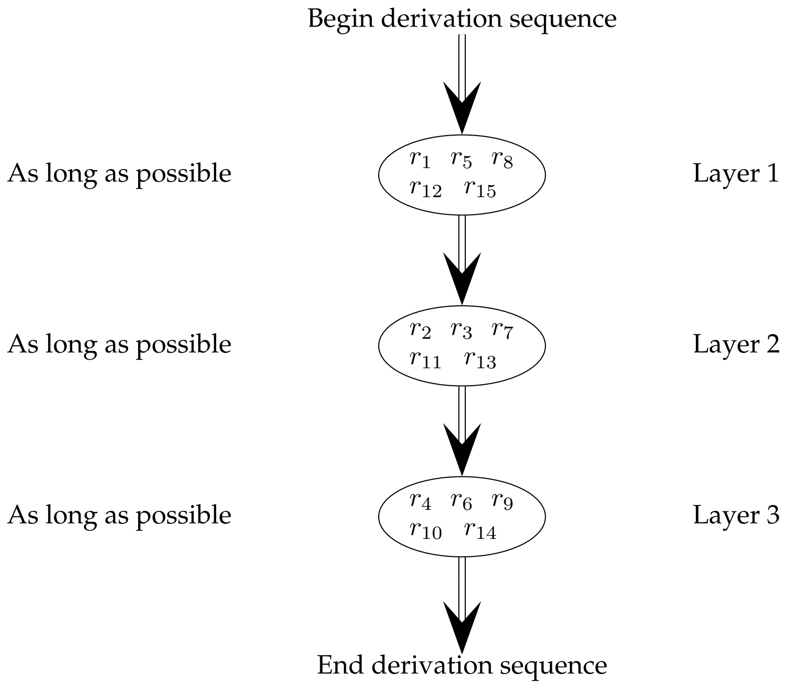

Begin derivation sequence
As long as possible
r_1 r_5 r_8 r_{12} r_{15}
Layer 1
As long as possible
r_2 r_3 r_7 r_{11} r_{13}
Layer 2
As long as possible
r_4 r_6 r_9 r_{10} r_{14}
Layer 3
End derivation sequence

Figure 3.6. Ordering rules in layers

Layered graph grammars were introduced by Rekers and Schürr [RS97]. Our layers are an adapted version of theirs; we do not base the ordering on object labels, but rather on rule functionality.

3.3.2 Parallel Independence

Two alternative derivations that may occur in any order with the same result are called *parallel independent*. The following definition (with adapted notation) is given by Ehrig et al. [EHK+97]:

Definition 5 (Parallel Independence). *Let $r_1 : L_1 \rightarrow R_1$ with NAC N_1 and $r_2 : L_2 \rightarrow R_2$ with NAC N_2 be two rules that may both be applied in a graph G. Let d_1 be the derivation of r_1 via the match m_1 (we write $d_1 = (G \overset{r_1,m_1}{\Rightarrow} H_1)$), and let d_2 be the derivation of r_2 via the match m_2 (denoted $d_2 = (G \overset{r_2,m_2}{\Rightarrow} H_2)$). Then we say that d_2 is **weakly parallel independent** of d_1 if $m_2' = r_1^* \circ m_2 : L_2 \rightarrow H_1$*

is a match for r_2 that satisfies N_2. We say that d_1 and d_2 are ***parallel independent*** *if they are mutually weakly parallel independent.*

To form m'_2, two steps are taken. First the matching $m_2 : L_2 \to G$ is applied, giving the elements of L_2 an image in G. Next, the co-production $r_1^* : G \to H_1$ is applied, thus bringing into action the mechanism of r_1. If the so constructed m' is a match for r_2 that satisfies N_2 it is now possible to apply r_2 — in other words, the application of r_1 did not affect the applicability of r_2.

The following requirements ensure parallel independence between two derivations [EHK+97]:

(1) neither derivation deletes an object that is necessary for matching the other, and

(2) neither derivation establishes a context that is forbidden by a NAC of the other.

Note that the definition focuses on parallel independence between derivations, i.e. specific instances of rule applications. We are in fact interested in parallel independence between the rules themselves, which means that all possible derivations of two rules must be parallel independent. When this is the case, we say that we have *true parallelism* between the rules.

In our system, we will be concerned with parallel independence within each layer. Since the rules of a layer are matched at random, the outcome could possibly be non-deterministic. If the rules in a layer are all parallel independent of each other, it will not matter in which order they are matched, and the outcome will be the same in all cases.

3.3.3 Sequential Independence

While parallel independence is considered between alternative derivations, *sequential independence* deals with consecutive derivations. When a derivation d'_2 does not rely on another derivation d_1 to be applied before it, we say that d'_2 is *weakly sequentially independent* of d_1. In other words, d'_2 is not causally dependent on d_1, so it does not matter whether d_1 takes place before or after d'_2. The stronger notion of (non-weak) sequential independence is not needed in this work. This definition (with adapted notation) is given by Ehrig et al. [EHK+97]:

Definition 6 (Weak Sequential Independence)**.** *Let $r_1 : L_1 \to R_1$ with NAC N_1 and $r_2 : L_2 \to R_2$ with NAC N_2 be two rules that may both be* *applied in a graph G. Let d_1 be the derivation of r_1 via the match m_1 ($d_1 = (G \overset{r_1,m_1}{\Rightarrow} H_1)$), and let d_2' be the subsequent derivation of r_2 via the match m_2' ($d_2' = (H_1 \overset{r_2,m_2'}{\Rightarrow} X)$). Then we say that d_2' is* ***weakly sequentially independent*** *of d_1 if $m_2 = (r_1^*)^{-1} \circ m_2' : L_2 \to G$ is a match for r_2 that satisfies N_2.*

To construct m_2, first the matching $m_2' : L_2 \to H_1$ is applied, giving the elements of L_2 an image in H_1 (note that since m_2' is a match for r_2, it would be possible in this situation to apply d_2'). Next, the inverse of the co-production $r_1^* : G \to H_1$ "undoes" the action of the derivation d_1, going back to G. If the resulting m_2 is a match for r_2 that satisifies N_2, it is now possible to apply d_2' — i.e. the fact that d_1 has been undone does not affect the applicability of d_2'. In other words, it does not matter whether or not d_1 has been applied before d_2'.

The following requirements ensure that the derivation d_2' is weakly sequentially independent of d_1 [EHK+97]:

(1) the overlapping of the right-hand side of d_1 and the left-hand side of d_2' does not contain elements which are generated by d_1, and

(2) no context that is forbidden by a NAC of d_2' is destroyed by d_1.

The first requirement guarantees that d_2' does not rely on d_1 to generate elements, and the second makes sure that d_2' does not rely on d_1 to destroy elements; in other words, d_2' does not rely on d_1 in any way.

In our system, the rules in precedent levels must be weakly sequentially independent of rules in subsequent layers. In other words, if the layers L_1 – L_{k-1} have been passed through — i.e. rules in these layers no longer apply — rules in layer L_k should not make those rules apply again. If they did, the system would not be stable (cf definition 16) after one pass of the layers.

Chapter 4

Terminology

After the preceding chapters which presented the background theory about PKI and graph transformations, we are now ready to present the terminology of our system, as well as to define some concepts and properties that pertain to it.

A model simplifies reality in order to show a structure. In this case, we are concerned with a view of the world that consists of entities (users), their knowledge of keys, and the certificates they pass between each other. The certificates themselves contain specific information about entities' keys that the entities sign for each other. In the model, entities act by adding, modifying and removing knowledge, and by distributing, modifying and revoking certificates to and from each other. Although our focus is revocation, we must also include distribution in order to model the structures of certificate chains that the revocation should act upon, as well as the actions that are allowed on revoked certificates.

4.1 Entities

The agents in our system are called *entities*. Each entity has an attribute:

- *Name*: the identifier of the entity, shown in figure 4.1 as V and W.

4.2 Statements

We define the different types of statements in the system — *keypair knowledge*, *public-key knowledge*, and *certificates* (the term *statement* is used to encompass all three). Knowledge represents key information that an entity has access to, whereas a certificate is a signed statement about keys passed between entities.

- *Keypair Knowledge*: An entity's knowledge of their own keypair, i.e. a pair of private and public keys generated by the owner.
- *Public-Key Knowledge*: An entity's knowledge of the public key of another entity[1].
- *Certificate*: a signed statement, passed between two entities, binding a public key to an entity.

4.2.1 Certificate Attributes

Every certificate has a set of attributes associated with it. To keep the graphs as clutter-free as possible — and because our model is purely an abstraction — we have kept the number of attributes to a minimum. For reference, we show a comparison of the X.509 v3 certificate structure and our certificate model in figure 4.1 (for details on the graphical representation, see section 5.1).

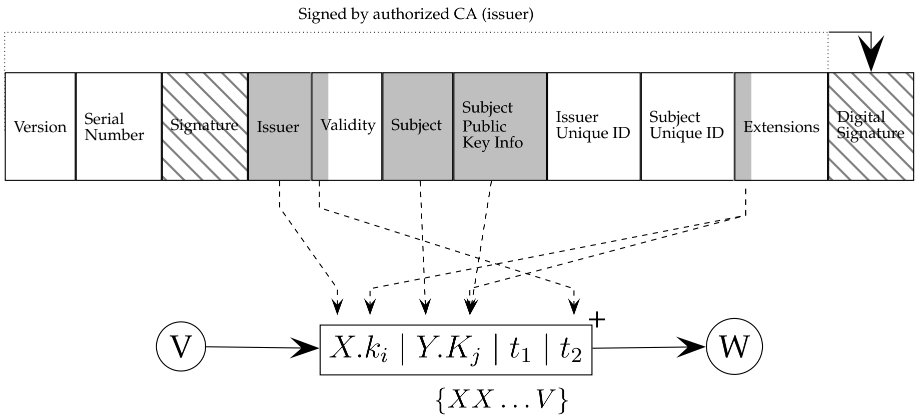


Figure 4.1. Comparison of X.509 version 3 certificate and our model

[1]Public-key knowledge can also be deduced knowledge about an entity's own public key, based on what other entities have told them.

The following attributes are used both in our model and in X.509:

- *Signer*: the identifier of the entity that owns the signing key, represented in figure 4.1 as X. In X.509, this entity is known as the "issuer".
- *Signing key*: the identifier of the private key used to sign a certificate, represented in figure 4.1 as k_i. We may also refer to this as the private key of the certificate, and denote private keys by lowercase letters, with an index to distinguish between multiple private keys of a signer. In X.509, the signing key can be identified by the extension "authority key identifier"; when not present, the signer is assumed to have only one signing key.
- *Subject*: the identifier of the entity being associated with the subject key, represented in figure 4.1 as Y. X.509 uses the same term.
- *Subject key*: the public key that is bound to the subject, along with its identifier, represented in figure 4.1 as K_j. We may also refer to this key as the public key of a certificate. Public keys are denoted by uppercase letters, with an index to distinguish between multiple keys of a subject. X.509 calls this the "subject public key info", and also uses the extension "subject key identifier" for multiple keys of a subject.

We also use some attributes which are not modelled in X.509: the *recipient*, *distributor*, *support set*, *state*, *subject time*, and *signing time* of a certificate. The first three of these are needed to keep explicit track of the paths on which certificates travel in the system; the fourth designates whether a certificate supports or denies a binding, or whether it has expired; the two last describe the time knowledge about a key was added to the graph (or modified), and the time that a certificate was signed, respectively.

- *Recipient*: the identifier of the entity receiving a certificate, represented in figure 4.1 as W.
- *Distributor*: the identifier of the entity passing/sending a certificate to the recipient, represented in figure 4.1 as V.
- *Support set*: a set of paths describing how information about the certificate has travelled in the system. The support set is placed below the certificate box. In figure 4.1, the support set contains one path,

beginning with the substring XX and ending with V. The formal definition of the support set can be found in section 4.3.1.

- *State*: a certificate is either *positive*, *inactive*, or *negative* — a positive certificate asserts the validity of the binding within it, whereas a negative certificate denies it. Inactive certificates are those where the public key has expired. The state is marked with a sign from the set $\{+, 0, 0^*, -\}$, and is placed at the upper right corner of the certificate box. The $+$ denotes a positive certificate, the 0 denotes an inactive certificate, and the $-$ denotes a negative certificate. The asterisk on the 0 is used to mark the inactive knowledge for a specific key that has the earliest inactivation time, out of the inactive knowledge for that key held by a given entity. This is necessary because when considering certificates signed with keys that have since become inactive, they should be accepted if they were signed before the earliest known inactivation time.

- *Subject time*: when knowledge about a key is added, inactivated, or negated, it is marked with the current time. As information about the key is spread via certificates, the time stamp stays the same. The main reason this field is necessary is to keep track of when a key was inactivated.

- *Signing time*: the time that a certificate was signed. X.509 uses the validity field to specify a time interval when a certificate is to be considered valid unless revoked — if an X.509 certificate is valid from the time of signing, the start of the validity period would be the same as our signing time. In our system, it is used as a comparison value to the subject time, when a key has been inactivated.

Two fields of the X.509 structure are implied in our model: the *signature (algorithm)* and the *digital signature*. The signature algorithm is assumed to be standardized in the system; the signature itself is assumed to have been created by signing the public key information with the signing key (using the algorithm).

When comparing a certificate σ to another certificate ρ, we say that σ is on the *same form* as ρ if both certificates have the same signer, signing key, subject and subject key. We say that σ is for the *same binding* as ρ if both certificates have the same subject and subject key.

4.2.2 Knowledge Attributes

Knowledge boxes also have a number of attributes, most of which are identical to those of certificates: subject, subject key, recipient/distributor (these are equal for knowledge boxes, and will also be referred to as the holder of the knowledge), support set, state, and subject time. Keypair knowledge has a private key as well, and no subject (since it is equal to the recipient and distributor). Public-key knowledge has a *type* attribute:

- *Type*: public-key knowledge is either of the types O or I, depending on whether it was created outside or inside the system. Type O knowledge represents those knowledge statements where the validity of the binding has been established by the distributor/recipient outside the system (in a secure, out-of-band procedure). Type I knowledge has been deduced from other knowledge and certificates.

The reason for keeping track of the type property is making sure that for each statement, there exists an entity that has verified the authenticity of the binding outside the system. To this end, we require that every type I knowledge be supported by at least one chain of other statements with either a type O public-key knowledge or a keypair knowledge at its root.

Type I knowledge boxes are drawn with single lines, while double boxes indicate type O. The terms *type I knowledge* and *deduced knowledge* may be used interchangeably, as may *type O knowledge* and *outside knowledge*.

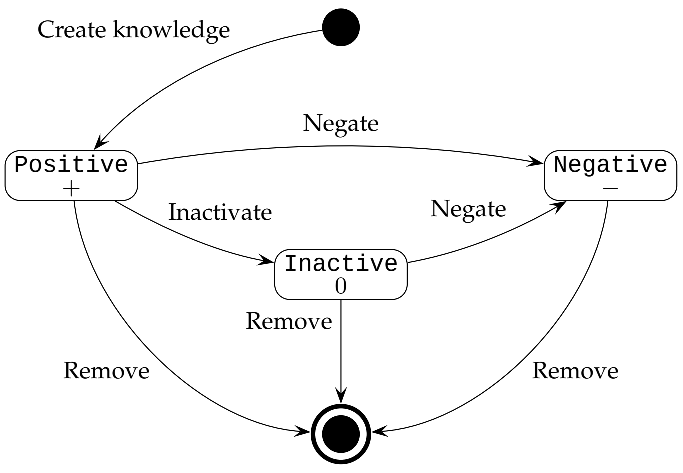


Figure 4.2. The states and transitions of knowledge

A UML diagram that shows possible state transitions for knowledge is shown in figure 4.2. For keypair and type O public-key knowledge, the transitions are induced by interactive participation by the holder of the knowledge. For type I knowledge, transitions will occur automatically, based on changes in the root knowledge for the supporting path. Note that once the knowledge leaves the positive or inactive state, there is way to make it positive again. This is a policy decision.

The transitions for certificates are naturally similar to those of knowledge, since they are based on knowledge held by an entity. When that knowledge changes its state, the state of depending certificates will change as well.

4.3 Support

The concept of *support* for knowledge and certificates is fundamental in our model. In order to know which statements are considered valid and allowed to remain in the system at any given time, we must keep track of the paths of certificates and deduced knowledge. At the root of each path, there must be keypair knowledge, or an outside public-key knowledge — the knowledge of an entity that has verified the binding of the subject and the subject key in a secure way. If this is the case, all the certificates and deduced knowledge in the paths that emanate from the root knowledge are considered valid. More formally:

Axiom 1 (Supported Knowledge)**.** *Keypair knowledge and type O public-key knowledge where the subject is an entity in the system is **supported**.*

In other words, keypair knowledge and outside public-key knowledge for existing entities is always considered to be supported. Certificates and type I public-key knowledge inherit their support via paths of quotations and deductions:

Definition 7 (Support for Certificates)**.** *A certificate π, signed by A with $A.k_i$, subject B, subject key $B.K_j$, subject time stamp t_b, certificate time stamp t_a, distributor C, and state s is **supported** if and only if:*

(1) If $C \neq A$:

(a) C is the recipient of a certificate ρ on the same form as π, and

(b) ρ has the state s and identical time stamps to those of π, and

(c) ρ is supported.

(2) If $C = A \neq B$:

(a) A has keypair knowledge σ of the keypair (k_i, K_i), which is either positive, or inactive with a time stamp $t < t_a$, and

(b) A has supported public-key knowledge ρ of $B.K_j$ with state s and time stamp t_b.

(3) If $C = A = B$:

(a) A has keypair knowledge σ of the keypair (k_i, K_i), which is either positive, or inactive with a time stamp $t < t_a$, and

(b) A has keypair knowledge of the keypair (k_j, K_j) with state s and time stamp t_b.

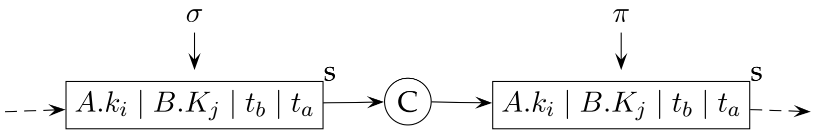

σ
π
s
s
$A.k_i \mid B.K_j \mid t_b \mid t_a$
C
$A.k_i \mid B.K_j \mid t_b \mid t_a$

Figure 4.3. Support for π, case 1

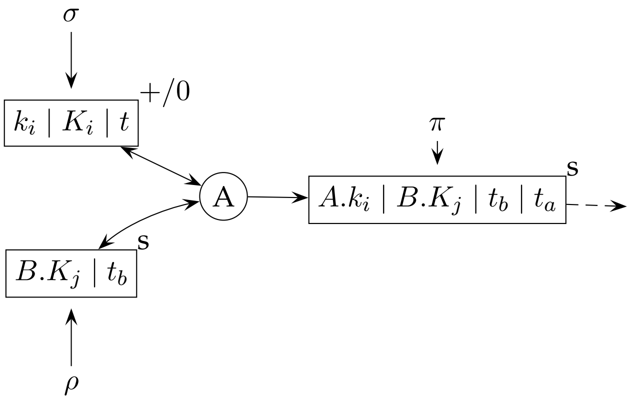


Figure 4.4. Support for π, case 2

The requirements of definition 7 are described in figures 4.3, 4.4, and 4.5 with the same notation as in the definition. The certificate π inherits its support from the statement σ, which may be either outside knowledge, deduced knowledge or another certificate.

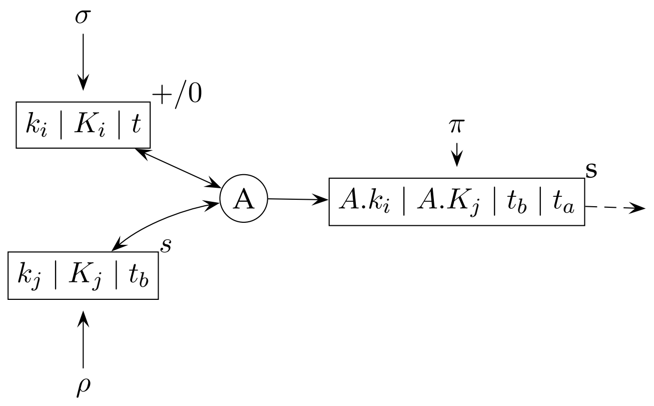


Figure 4.5. Support for π, case 3

Definition 8 (Support for Deduced Knowledge). *A deduced knowledge ρ with holder A, subject C and subject key C.K is **supported** if and only if A is the recipient of a certificate ϕ that:*

(1) certifies the same binding as ρ, and

(2) is signed with $B.k$ at time t_b, and

(3) has the same state s as ρ, and if either

(4) (a) A has positive, supported knowledge μ stating that they recognize $B.K$ as B's public key, or

(b) A has inactive, supported knowledge μ with a time stamp $t > t_b$ stating that they recognize $B.K$ as B's public key, and no inactive, supported knowledge for the same binding as μ with an earlier time stamp.

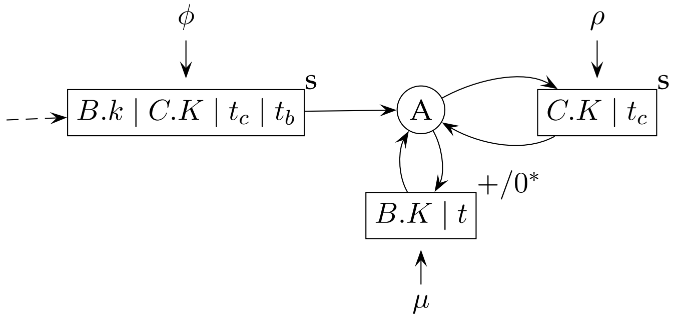


Figure 4.6. The deduced knowledge ρ is supported if ϕ and μ are supported

The requirements of definition 8 are described in figure 4.6. Two supporting statements for ρ are required: an incoming certificate ϕ that supplies the information about a binding, as well as the knowledge μ, which gives A the means to verify the signature in ϕ. The knowledge μ may be either a deduced knowledge or an outside knowledge, as long as it is supported.

To summarize, support paths can consist of both quoted certificates and deduced knowledge.

4.3.1 Support Sets

In order to keep track of the support of statements in the system, every knowledge and certificate has a support set. The support set consists of all acyclic paths that connect the given statement to a type O knowledge, or a keypair knowledge. This is a way to control which knowledge is the root of other statements in a path.

Definition 9 (Valid Path)**.** *A **path** is a string over the set of entity names. A **valid path** of a given statement σ is a path representing an acyclic supported — as defined in definitions 7 and 8 — chain of statements from a type O knowledge or a keypair knowledge via quoted certificates and deductions to σ.*

The reason we need to keep track of valid paths is the cycle problem described in section 1.2.1. Only acyclic paths should be considered as support for a given statement.

Definition 10 (Support Set)**.** *A **support set** is a set of paths. The support set of a given statement σ is **valid** if it contains only valid paths of σ; it is **complete** if it contains all valid paths of σ.*

Paths are denoted by lower-case Greek letters, and for support sets we use upper-case Greek letters. As a shorthand, concatenation of a support set and a user name represents the operation of concatenating all paths in the set with the user name. If the support set $\Pi = \{\pi_1 \ldots \pi_i\}$, then $\Pi A = \{\pi_1 A, \pi_2 A, \ldots, \pi_i A\}$.

Figure 4.7 shows a small graph with support sets below the statements (time stamps are omitted for clarity). Root knowledge (type O public-key knowledge, or keypair knowledge) always carries the name of the holder as the only path of the support set, as they are always the very beginning of a path. Subsequent quotations and deductions concatenate the distributor's name to the existing support set, provided that no cycle was formed in the latest step. Cycles can be detected by comparing the support sets of an incoming edge E and outgoing edges that are already in place: if there

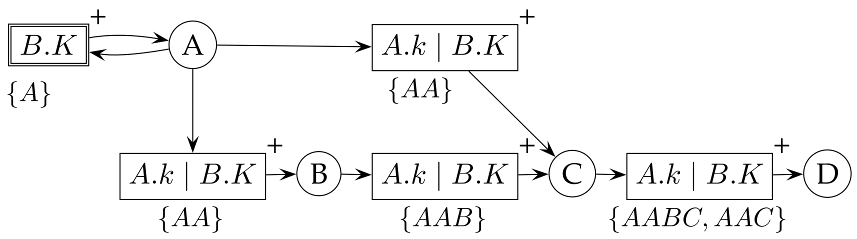


Figure 4.7. Support sets

is no path in the support sets of outgoing edges that is a prefix of one of the paths in the support set of E, then no cycle has formed. When two or more paths meet — as when C receives two certificates on the same form in figure 4.7 — they all become part of the support set.

4.4 Collisions

Since we allow positive statements as well as inactive and negative statements in our system, it is possible that entities hold contradictory knowledge about a binding. When this happens, we say that there is a *collision*. There are two types of collisions: *inactive* and *negative collisions*.

Definition 11 (Inactive Collision). *Given a knowledge σ which is positive, and a knowledge ρ which is inactive, both for the binding between B and $B.K_j$, there is an* ***inactive collision*** *if they have the same recipient C.*

Definition 12 (Negative Collision). *Given a knowledge σ which is positive or inactive, and a knowledge ρ which is negative, both for the binding between B and $B.K_j$, there is a* ***negative collision*** *if they have the same recipient C.*

Collisions can be handled in different ways, e.g. by letting either positive or negative knowledge take precedence, or by some other policy. We have chosen to give precedence to inactive knowledge when there are no negatives present, and precedence to negative knowledge when present. In this way we get an ordering relation between the three states: accept positive knowledge if there is no inactive or negative present; accept inactive knowledge if there are no negatives present; always accept negative knowledge. The rationale for this choice is the following: Since we assume that all entities are trustworthy, if someone has issued an inactive or negative statement for a binding, then there must be a reason why that binding should not be considered valid. It is safer to believe in the revocation statement than in the asserting statement.

Note that we only consider contradictory knowledge statements to be in conflict when they are held by the same entity. We do not handle the case of different entities holding contradictory knowledge, since this can be considered to be a natural situation in a distributed system. Furthermore, as described in section 4.6, entities are assumed to have access only to their own information — they would not be aware of other entities holding contradictory knowledge. If total control the entities' information is desired, it may be better to use a system with a central CA, or to implement another policy on the spreading of inactive and negative statements.

4.5 Valid State

We have now discussed all the concepts we need to define what we mean by a *valid state* for a certificate system.

Definition 13 (Valid State)**.** *A system is in a* ***valid state*** *if and only if:*

(1) the support set of each statement is valid, complete and non-empty;

(2) there are no collisions;

(3) where there is support for deduced knowledge, the deduction is made;

(4) when an entity has one or more inactive public-key knowledge statements about a specific binding, the one with the earliest inactivation time is marked.

The first item requires that all statements are supported. If the support set is valid and complete, then it contains all valid paths. If it is non-empty, then there is at least one such valid path, i.e., there is support for the certificate. The second item requires no collisions, i.e., in a valid system there is no entity that holds both positive/inactive and negative knowledge about the same binding. The third item requires that deduced knowledge is generated when the support is present. If an entity has the necessary information to draw a conclusion, it is logical that it should be drawn, in the spirit of modus ponens. The fourth item requires the earliest inactivated knowledge that an entity holds for each binding to be marked. The reason for this is that when basing decisions on inactive knowledge, the earliest inactivation time should be considered.

Note that a system may not always be in a valid state. The moment a negative certificate is distributed to an entity that already has a positive certificate for the same binding, there is a collision and hence the second requirement is violated. The system should be able to handle such situations and be able to reach a valid state again. In chapter 6 we will analyze the rules of our system with respect to the notion of validity.

4.6 Assumptions

Our model is global in the sense that all statements which have been distributed in the system are shown in a single graph. It is local, however, in the sense that each entity is aware of only those statements which it has itself generated, received or distributed. We state the following assumptions about what information each entity has access to:

Assumption 1 (Local Awareness). *An entity is aware of all and only those statements which it is currently receiving and distributing.*

In other words, each entity has a local view of the system graph, and can only see incoming and outgoing edges. Since there is no history of which statements have previously been present, an entity can only be aware of the current state.

Assumption 2 (Local Decisions). *An entity can base its decisions only on those statements of which it is aware.*

Since an entity only has a local view of the system graph, it cannot use information from other parts of the graph.

Assumption 3 (Local Changes). *An entity can directly affect only those statements of which it is aware.*

Note that the effect of changing a statement that an entity is distributing may propagate in the paths that stem from that statement, although the entity only can directly affect the first statement of those paths. Changes in those statements which an entity is receiving are only intended to be performed via deductive rules, i.e. automatic rules which are executed by the system to keep the graph in a valid state — see chapter 5 for details.

Entities are assumed to store their knowledge and certificates securely — only by applying one of the rules in chapter 5 can information be spread in the system.

Chapter 5

Modeling Key Certificates

In this chapter we will present the C-graph formalism, which is a model describing entities of a system and their knowledge of keys and of public key certificates. By adding graph transformation rules that act on a C-graph, we can breathe life into the model and allow the entities to issue, distribute and revoke the certificates they have access to.

5.1 The C-Graph

The formalization we will use to model a distributed PKI is a directed, attributed graph with attributes taken from the predefined sets V-ATT and E-ATT. We call such a graph a *C-graph*.

Definition 14 (C-Graph). *A C-graph is an attributed graph* $(V, E, s, t, \textit{v-att}, \textit{e-att})$ *where* V *is the set of entities in the system and* $E = E_P \cup E_K \cup E_C$ *is the set of (entities') public-key pairs, their knowledge of public keys, and their signed statements about (entities') public keys. The functions* $s, t : E \to V$ *assign a distributor and recipient to each edge, respectively, and v-att* $: V \to$ *V-ATT and e-att* $: E \to$ *E-ATT assign attributes. The node attribute function v-att assigns names from a name alphabet V-ATT to nodes. For simplicity of presentation, we split the edge attribute function e-att into three — e-att*$_K$*, e-att*$_P$*, and e-att*$_C$*, one for each type of edge — with several subfunctions:*

$$\textit{e-att}_K = \begin{cases} \textit{keypair}: & E \to (ID_k \times k) \times (ID_K \times K) \times T \\ \textit{state}: & E \to \{+, 0, -\} \\ \textit{support}: & E \to \textit{V-ATT} \end{cases}$$

$$\textit{e-att}_P = \begin{cases} \textit{public-key}: & E \rightarrow (ID_K \times K) \times T \\ \textit{type}: & E \rightarrow \{O, I\} \\ \textit{state}: & E \rightarrow \{+, 0, 0^*, -\} \\ \textit{support}: & E \rightarrow 2^{\textit{V-ATT}^*} \end{cases}$$

$$\textit{e-att}_C = \begin{cases} \textit{cert}: & E \rightarrow ID_k \times (ID_K \times K) \times T^2 \\ \textit{state}: & E \rightarrow \{+, 0, -\} \\ \textit{support}: & E \rightarrow 2^{\textit{V-ATT}^*} \end{cases}$$

The subfunctions of the edge attribute functions are defined separately. For $\textit{e-att}_K$, the attribute function for entities' knowledge of their own keypairs, *keypair* assigns a private and a public key, along with the ID for the keypair (private and public keys of the same key pair are assumed to share an ID), and a time stamp indicating the time of creation to the edge. The *state* function describes if the knowledge is positive, inactive or negative, and *support* assigns a support set. In the case of keypair knowledge, the support set is just the name of the key owner, hence a single member of *V-ATT* is sufficient.

The attribute function $\textit{e-att}_P$ describes entities' knowledge of other entities' public keys, whether obtained outside the system, or deduced within the system. The *public-key* function assigns a public key with corresponding ID, along with a time stamp indicating the time the knowledge was formed. The *type* function distinguishes between knowledge obtained from the outside and knowledge deduced within the system, and the *state* function describes if the knowledge is positive, inactive, or negative, with the 0^* state used to designate the earliest inactivation time for a specific key among the knowledge possessed by each entity. The *support* function assigns a support set (a set of strings over the set of entity names) to each knowledge.

For $\textit{e-att}_C$, the attribute function for certificates, the certificate function *cert* assigns a private key ID (of the signing entity) and a public key ID and key (of the subject entity), along with two time stamps, as attributes to each edge — knowing which signing key was used gives entities enough information to verify the signature, and knowing the ID of the public key as well as the public key itself gives entities information to use the public key in new verifications. The time stamps specify when knowledge of the public key was formed or inactivated, and when the certificate was created. The reason that two time stamps are needed is that entities need to know both the time that the public key was inactivated (if it was), and when the certificate itself was created (in case the signing key itself has been inactivated). The functions *state* and *support* are similar to the corresponding functions in $e\text{-}att_P$.

5.1.1 Graphical Representation

Attributes are represented in different ways graphically. The knowledge or certificate itself (the *keypair*, *public-key* or *cert* function) is represented by a box on each edge, holding the attributes.

Keypair knowledge boxes contain the private key identifier at left, and the public key identifier in the middle (the actual keys are implied). The time stamp is at right, and indicates the time the keypair was created, or inactivated (when applicable). Since keypair knowledge always belongs to the owner of the keypair, it is not necessary to state the name of the key owner in the box.

Public-key knowledge boxes contain the identifier of the public key that the distributor/recipient has knowledge of (the actual key is implied). The time stamp is at the right. If the public-key knowledge is of type O, the time stamp indicates the time that the knowledge was formed (or inactivated, when applicable); if the knowledge is of type I, the time stamp indicates the time stamp of the root of the path leading to that knowledge. Edges representing O knowledge are shown with double boxes; edges representing I knowledge are shown with single boxes.

Certificates contain first the signing key identifier, and then the identifier of the public key being certified (the actual key and the signature are implied). The two rightmost elements are time stamps. The first one indicates the time that the knowledge of the public key (at the root of the path leading to the certificate) was created or inactivated (when applicable); the second one indicates the time that the certificate itself was created.

The value of the *state* function is shown at the top right corner of each box. Positive knowledge or certificates are marked with a plus sign, inactive knowledge or certificates are marked with a zero (possibly with an asterisk), and negative knowledge or certificates are marked with a minus sign. Finally, the (valid and complete) *support set* of a knowledge or certificate is shown below the box.

An entity can have any number of private/public key pairs associated with it, easily distinguished with an index; the public keys of entity A can be numbered $A.K_1$, $A.K_2$, etc. In the following rules we may sometimes leave out this index to simplify the notation, but note that in a matching of $A.k$ and $A.K$, the private and public keys are assumed to have the same index.

5.1.2 Examples

Figure 5.1 shows A's keypair knowledge of (k_i, K_i), created at time t. The keys in a keypair knowledge are written without the key owner's name, since the owner is always the holder of the knowledge.

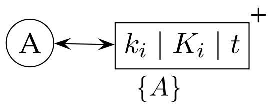


Figure 5.1. A has the keypair (k_i, K_i)

In figure 5.2 there are two entities, A and B. The edge represents A's public-key knowledge of $B.K_j$, which was obtained at time t. The double lines marking the box implicate that the knowledge has been obtained outside the graph system (A may have been told by B in person). The support set — below the certificate box — has only one member, the path A. O certificates are always the first edges on a path, and are always marked with the signer as the path attribute. For clarity, we have included the node representing B in this figure. In subsequent figures, the signer and subject are implied when they are not identical to the distributor or recipient.

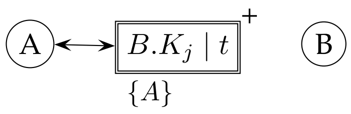


Figure 5.2. A has knowledge of $B.K_j$

Figure 5.3 describes a certificate which is quoted (distributed) by C and received by D. The box on the edge represents a certificate for $B.K_j$, signed by A at time t_2. A's knowledge of B was formed at time t_1. The path attribute $\Pi = \{\pi_1, \ldots, \pi_k\}$ represents the valid and complete support set of the certificate.

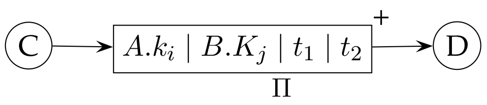


Figure 5.3. C quoting a certificate to D

5.2 Rules for the C-Graph

By specifying a set of graph transformation rules that act on a C-graph, we can model a system for certificate distribution and revocation. These rules are either functions that the users can call if certain conditions are met — *interactive rules* — or automatically applied deductions — *deductive rules*. Deductive rules bring the graph to a valid state (e.g. by deriving new knowledge or removing unsupported knowledge) after an interactive rule has been applied. In interactive rules, variables are from the beginning of the alphabet; deductive rules have variables from the end.

New outside knowledge and certificates are added with a time stamp value t_c, which is intended as the current global time in the system. We assume an ordering relation between time values such that $t_1 < t_2$ if t_1 took place before t_2.

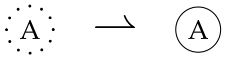
When a rule describes an edge with a single box, it is still possible that it is matched into a double-edge box. Double-edge boxes can be seen as a special case of single-edge boxes, and the main reason they are present are to act as a visual guide to which certificates have originated outside (note that a support set with a single member is sufficient to indicate outside knowledge). When double-edge boxes are specified in a rule, they can only be matched into double-edge boxes in the host graph.

Note that the rules do not describe the protocol for exchanging information. For example, in the rule for quotation (rule 7), we do not know why D chooses to quote the certificate to E. E may have requested the information from D, but the mechanics of that request are outside the scope of this work.

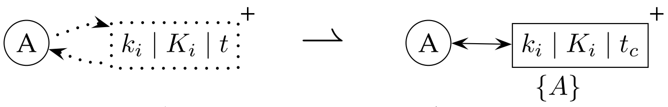
We will present the rules in steps: first the rules that add knowledge and certificates to the graph, then three different kinds of revocation — through removal, inactivation, and negation (we refer to section 6.1 for a flowchart representation of the revocation functionality described in this chapter). Rules are presented in framed boxes to clearly distinguish them from examples. An example will run throughout the chapter, with changes made to the example graph after each section of presented rules.

5.3 Adding Entities, Knowledge, and Certificates

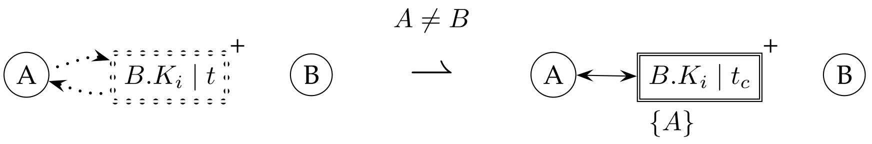
The first five rules describe the addition of entities, knowledge, and certificates to the system.



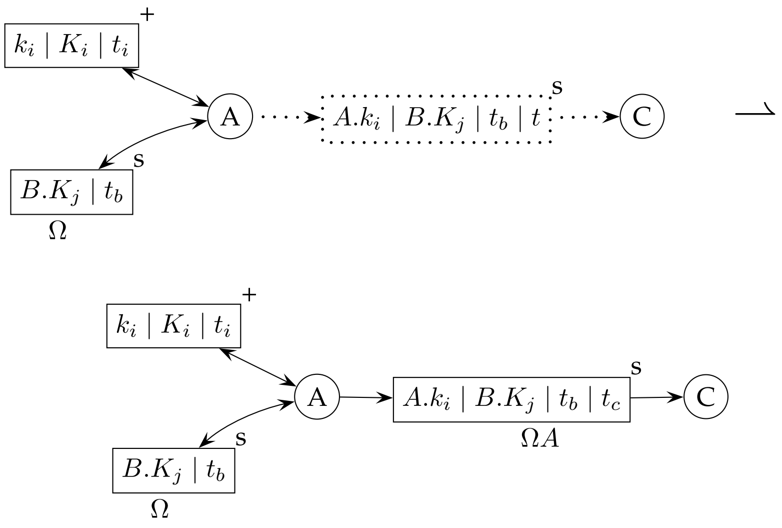
Rule 1: Adding an entity



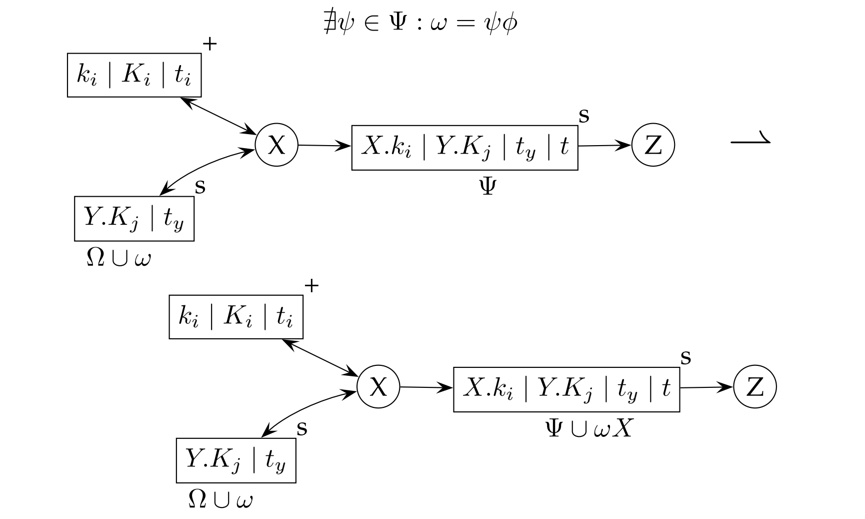
Rule 2: Generating a new key pair



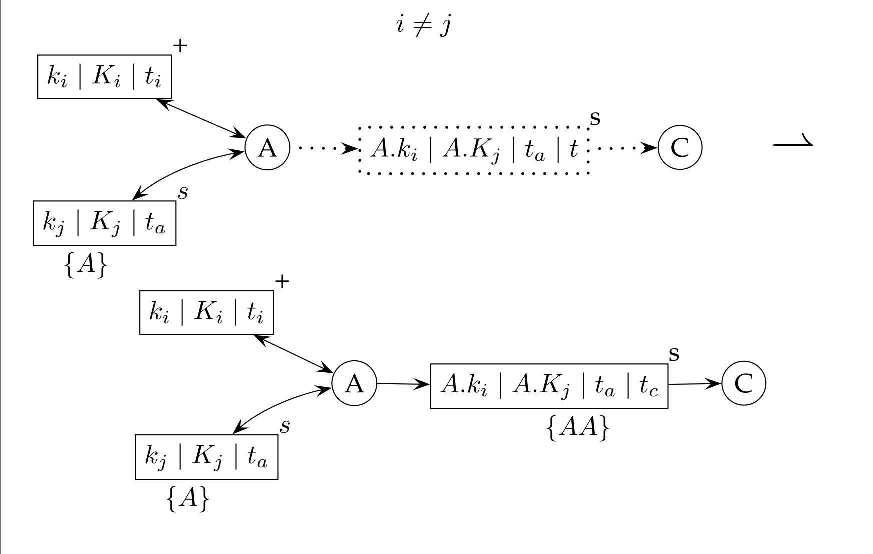
Rule 3: Adding outside knowledge


$k_i \mid K_i \mid t_i$
$+$
A
$A.k_i \mid B.K_j \mid t_b \mid t$
s
C
$B.K_j \mid t_b$
Ω
$A.k_i \mid B.K_j \mid t_b \mid t_c$
ΩA

Rule 4: Creating a certificate



Rule 5: Updating the support set of a certificate



Rule 6: Creating a certificate for one’s own key

Rule 1 allows the addition of entities to the system. This makes it something of a meta-rule, and is intended to be called by an administrator outside the system. Rule 1 is the only administrative rule.

The interactive rule 2 allows entities to add new key pairs of their own, and is called interactively by the key owner A. The knowledge will be stamped with the current time t_c, and the NAC avoids duplicates.

Rule 3 (called by A) lets entities add knowledge interactively that they have acquired outside the system. The NAC avoids duplicates.

Rule 4 allows entities to interactively create certificates for keys that they know, using one of their own keys — that they believe to be non-revoked — to sign the certificate. Duplicates are avoided through the NAC. Note that the certificate inherits its support set and the time stamp for $B.K_j$ from A's knowledge of $B.K_j$.

Rule 5 is a deductive rule, which ensures that the support set of the first certificate in a chain (i.e. a certificate created through the use of rule 4) is kept updated, when there is added support. The rule adds all existing paths to Ψ (the support set of the certificate), while excluding cyclic paths through the condition that there must not be a path in Ψ that is a prefix of ω, the path to be added.

Rule 6 lets entities interactively create certificates for their own keys, using another key than the certified one to sign the certificate. There is no need for a rule that updates the support for such certificates, since they are based on the keypair knowledge of A, which has the non-changing support set $\{A\}$. The NAC constraints are similar to those in rule 4.

5.3.1 Example

We will now give an example to illustrate the use of the rules for adding entities, outside knowledge, and certificates. Starting with an empty system, the administrator applies rule 1 five times to create entities A, B, C, D and E. The negative application condition of the rule ensures that each entity has a unique name. The result is shown in figure 5.4.

The new entities now start acting. A creates the key pair (k_i, K_i), B creates the key pair (k_j, K_j), and D creates the key pair (k_l, K_l), each using rule 2 to add them to the system. A has outside knowledge of $B.K_j$, and D has outside knowledge of $A.K_i$. They each apply rule 3 to add their outside knowledge to the system. The NACs of both rules ensure that the added items are not duplicates. The result of these operations is shown in figure 5.5.

Next, A decides to tell B and C about their knowledge of $B.K_j$, using rule 4 to create certificates and sign them with $A.k_i$. After these operations, the system will look as shown in figure 5.6.

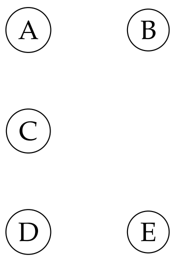


Figure 5.4. New entities

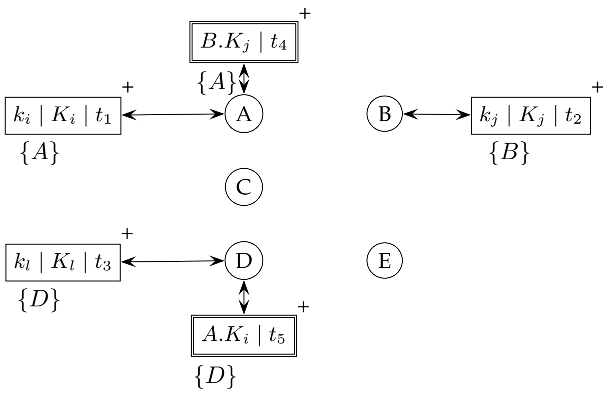


Figure 5.5. Outside knowledge added by A and D

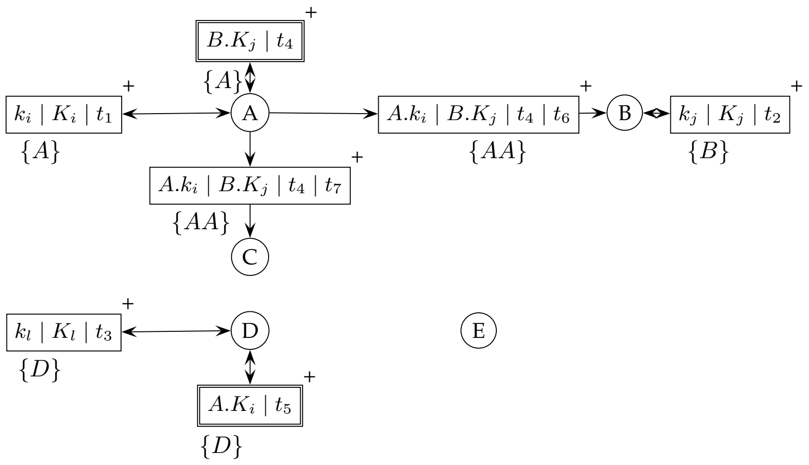
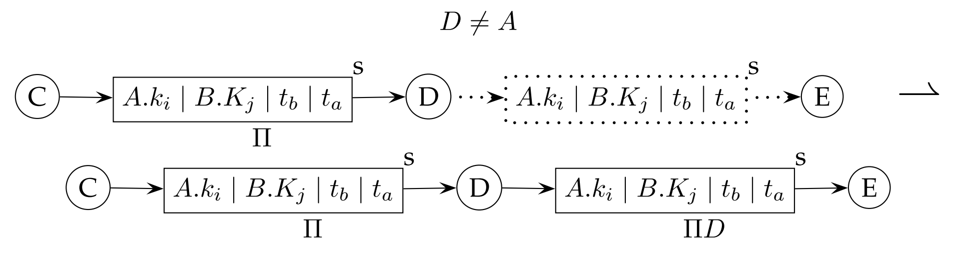


Figure 5.6. Certificates added by A

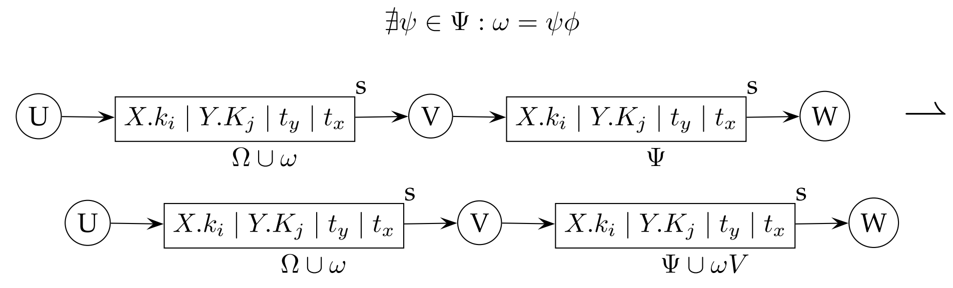
5.4 Quotation

Certificates can be quoted, as described by the interactive rule 7: if D has been told by C about A's certification of $B.K_j$, D can spread that information to E. Since the state of the quoted certificate is given as a variable s, the rule describes quotation of positive certificates as well as inactive and negative ones. The state of a quoted certificate is the same as the one that supports it (cf. the matching condition in section 3.2).

Rule 8 is deductive, and updates the support set of quotations when necessary. The rule applies to quotations of all three states.



Rule 7: Quotation


$\nexists \psi \in \Psi : \omega = \psi\phi$
U
$X.k_i \mid Y.K_j \mid t_y \mid t_x$
s
$\Omega \cup \omega$
V
W
Ψ
$\rightharpoonup$
$\Psi \cup \omega V$

Rule 8: Updating the support set

5.4.1 Example

C now quotes to D the certificate that they received from A. To add the quotation, rule 7 is applied by C. The NAC of the rule checks for duplicate quotations. Note how the support sets are formed, indicating the path back to A's outside knowledge of $B.K_j$. The result is shown in figure 5.7.

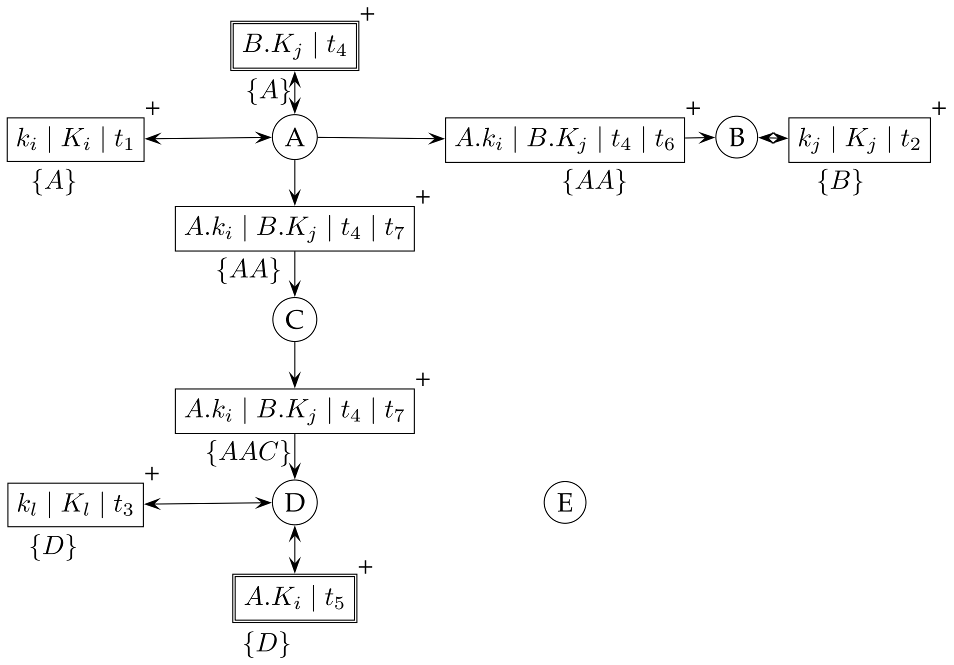


Figure 5.7. Quotation added by C

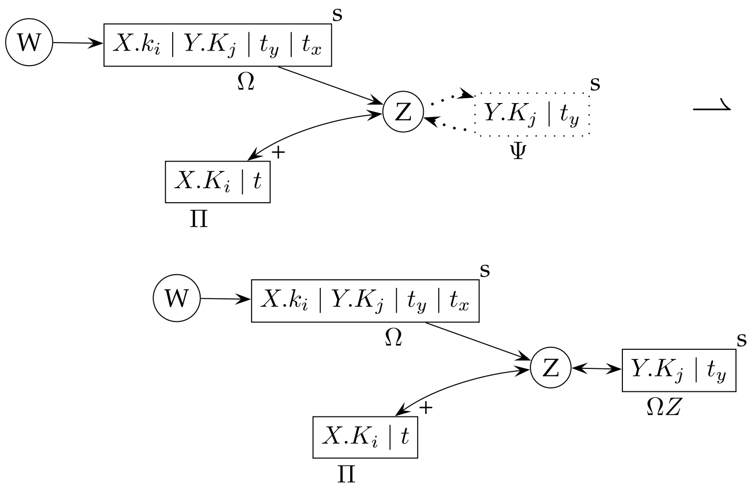
At this point the system is not in a valid state, because D is able to make a deduction about $B.K_j$. Rules for deducing new knowledge are presented in the following section, so the example will be suspended until section 5.5.1, where the deduction will be carried through.

5.5 Deducing Knowledge

The deductive rules 9, 10, 11, and 12 ensure that new knowledge is deduced when there is support for it, and that the support sets of such knowledge are kept updated. The rules apply to deductions with any state, but note that the variable s must be instantiated to the same state when it appears several times in the same rule.

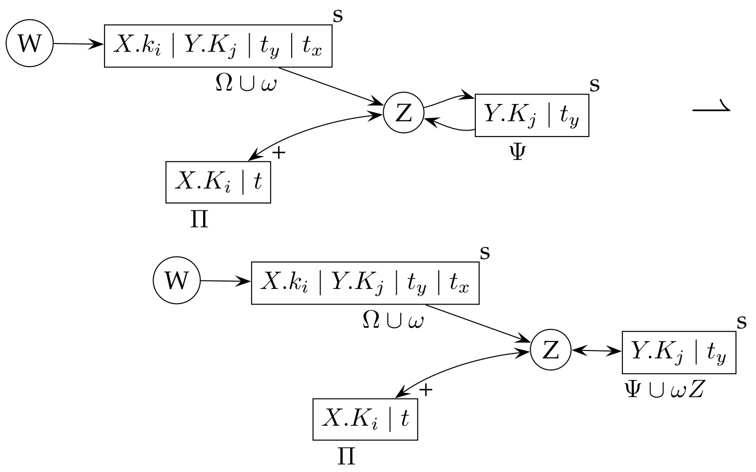
Rule 9 works as follows: if Z has knowledge ρ that $X.K_i$ is X's public key, and a certificate σ signed with $X.k_i$ for $Y.K_j$, Z has the ability to verify the signature on σ and will deduce type I knowledge about $Y.K_j$. The

$$\Psi \neq \{Z\}$$

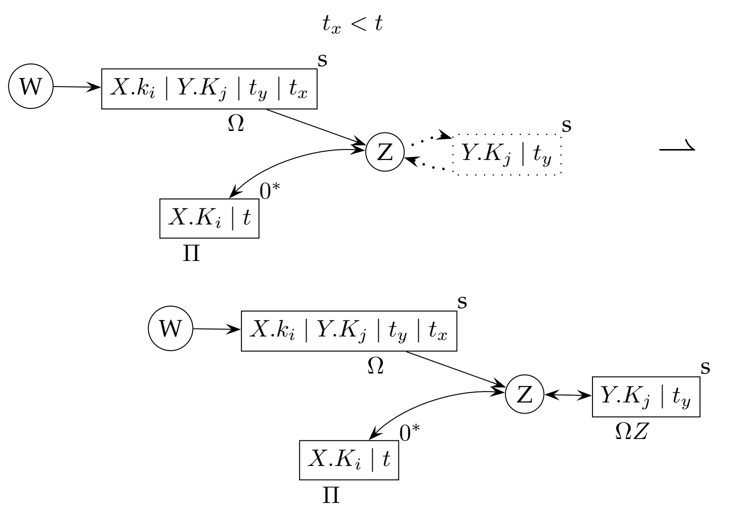


Rule 9: Deducing new knowledge

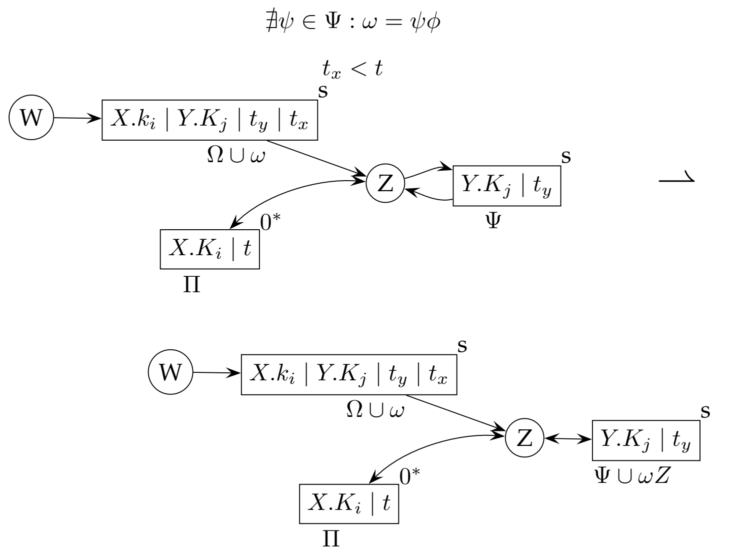
$$\nexists \psi \in \Psi : \omega = \psi\phi$$



Rule 10: Updating the support set of deduced knowledge



Rule 11: Deducing new knowledge with inactive signing key



Rule 12: Updating the support set of deduced knowledge

support set of the new knowledge is derived from the support set of σ. When new certificates are given to Z on the same form as σ, rule 10 will add that support to the support set of Z's knowledge, similarly to rule 8. Rules 11 and 12 work correspondingly, in the case that ρ is inactive. In this case, the time stamps of ρ and σ must be compared: σ can only be accepted if it was created before the earliest known inactivation of ρ.

5.5.1 Example

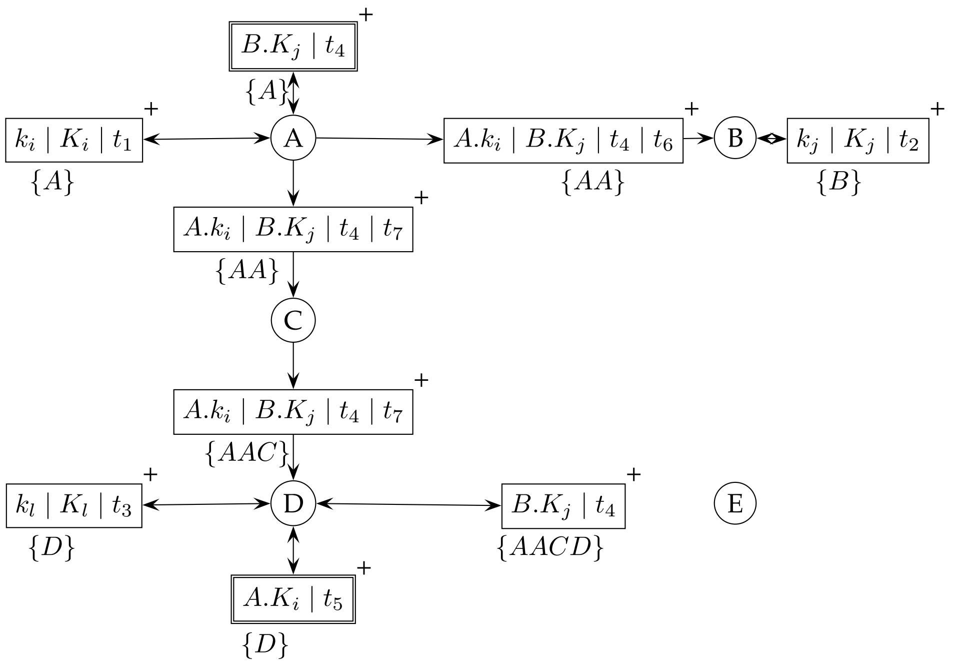


Figure 5.8. Deduction made by D

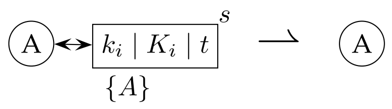
When the quotation from C to D was added (in section 5.4.1), the deductive rule 9 applies: D has knowledge about $A.K_i$, as well as a certificate signed with that key. Therefore, D has the possibility to check A's signature, and deduces knowledge about $B.K_j$. The checking of the signature may be performed at any time; if the signature turns out not to be correct, D may choose to apply some form of revocation. Note how the support set as well as the time stamp of the deduction trace back to A's original knowledge. The result of this automatic process is shown in figure 5.8.

5.6 Revocation by Removal

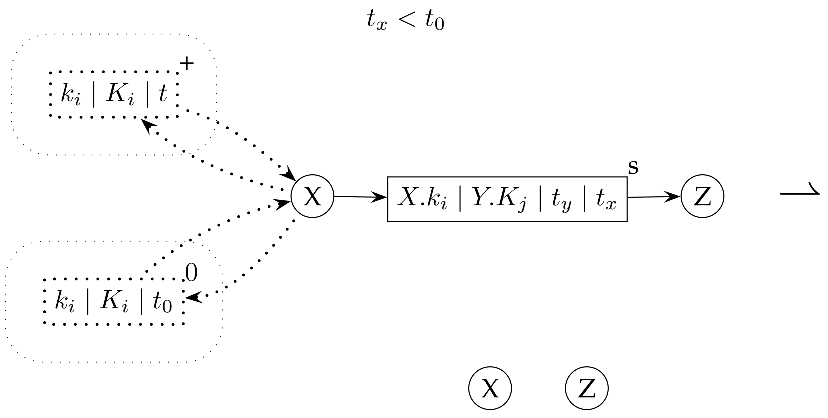
Removal of knowledge or certificates should be interpreted as the revoking user no longer being able to vouch for the authenticity of the statement in question. We will now describe how such removal affects the graph. For clarity, the interactive and deductive rules for removal will be presented in sections describing removal of keypair knowledge, public-key knowledge, and certificates, respectively.

The rules in this section all apply to knowledge or certificates of any state s.

5.6.1 Removal of Keypair Knowledge

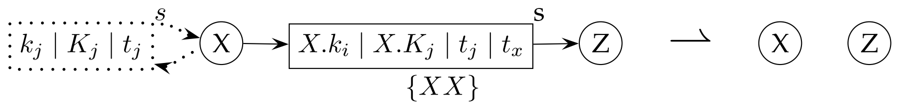


Rule 13: Removing keypair knowledge



Rule 14: Invalid signature

Rule 13 is an interactive rule that lets entities remove keypair knowledge that they hold. Rule 14 is deductive, and removes certificates with

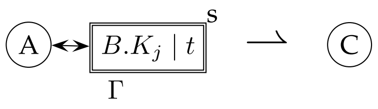


Rule 15: Removal of certificate without support

signatures that have become invalid due to removal (or inactivation, see section 5.7) of keypair knowledge. There are two NACs; the top NAC describes the lack of a positive keypair knowledge, and the bottom NAC describes the lack of inactive keypair knowledge that was inactivated after the public-key knowledge of $Y.K_j$ was formed. If neither of the two keypair knowledge variants are present, the signature cannot be accepted, and the certificate must be revoked. Note that Y may be matched into the same entity as X, for certificates that are signed by the key owner.

Rule 15 is also deductive, and removes certificates for a distributor's own keys, when there is no longer corresponding keypair knowledge of the same state.

5.6.2 Removal of Public-Key Knowledge

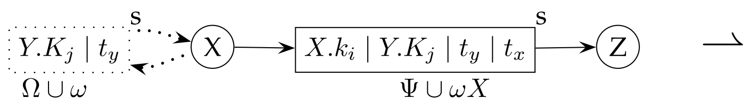


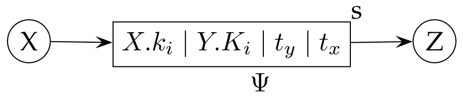
Rule 16: Removing type O public-key knowledge

The interactive rule 16 lets entities remove type O public-key knowledge that they hold. It is unnecessary to remove deduced knowledge, since it will reappear automatically as long as the premises for it are present.

Rule 17 is deductive, and updates the support set of the first certificate in a chain when some support is missing. If there is no support left, rule 21 will remove the certificate itself.

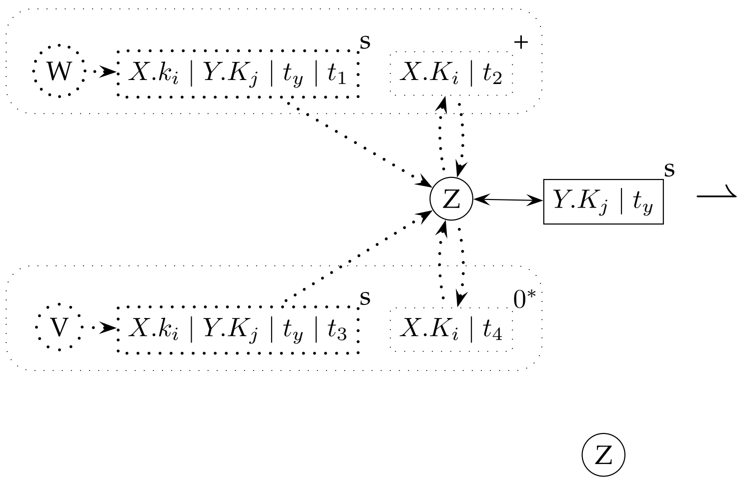
The deductive rule 18 removes deduced public-key knowledge that is no longer supported, either because there is no incoming quotation with a





Rule 17: Updating the support set

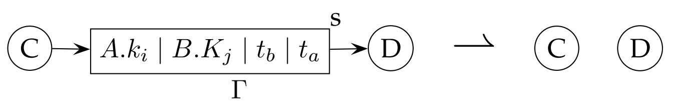
$t_3 < t_4$


W
$X.k_i \mid Y.K_j \mid t_y \mid t_1$
s
$X.K_i \mid t_2$
+
Z
$Y.K_j \mid t_y$
s
V
$X.k_i \mid Y.K_j \mid t_y \mid t_3$
s
$X.K_i \mid t_4$
0*
Z

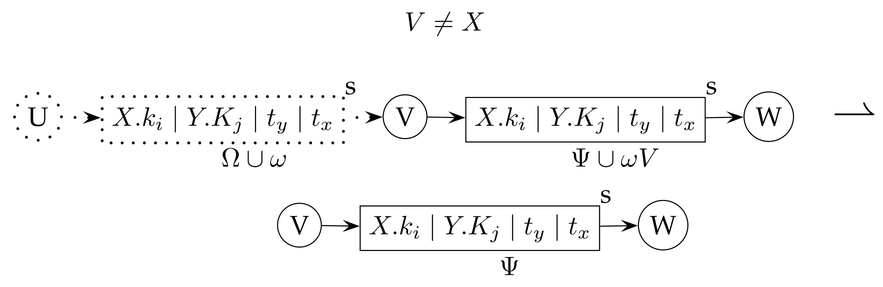
Rule 18: Removal of deduced knowledge

matching positive public-key knowledge, or because there is no incoming quotation with a matching inactive public-key knowledge (where the inactivation took place after the quoted certificate was signed). Note that the constraints of rule 18 handles all three possible scenarios of lost support: quotation present but knowledge revoked, quotation revoked but knowledge present, or both quotation and knowledge revoked. Remember that within a constraint that consists of several elements, all elements must be present for the constraint to be violated. If both elements within one of the constraints are present there is support for the deduced knowledge, and indeed the rule should not apply.

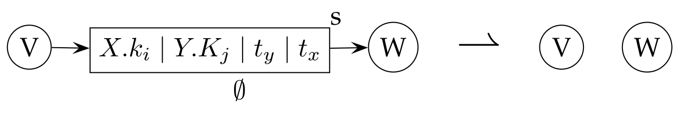
5.6.3 Removal of Certificates



Rule 19: Removing a certificate


$V \neq X$
U
$X.k_i \mid Y.K_j \mid t_y \mid t_x$
s
$\Omega \cup \omega$
V
$X.k_i \mid Y.K_j \mid t_y \mid t_x$
s
$\Psi \cup \omega V$
W
V
$X.k_i \mid Y.K_j \mid t_y \mid t_x$
s
Ψ
W

Rule 20: Updating the support set

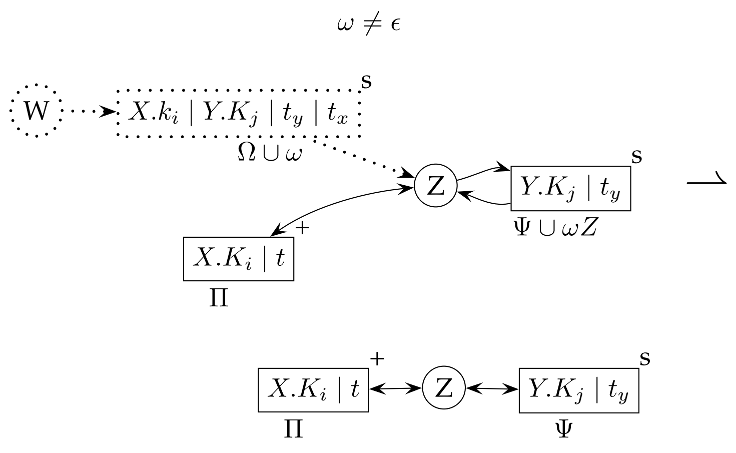


Rule 21: Removal of unsupported quotations

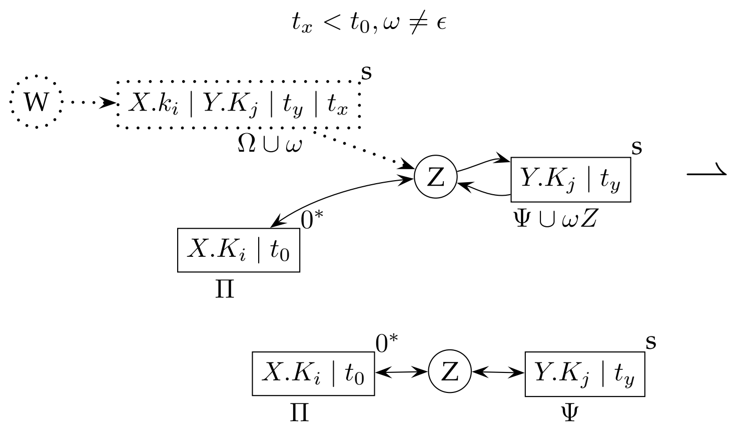
The interactive rule 19 lets entities remove certificates of any state that they have distributed. The rest of the rules in this section are deductive.

Removing a certificate can affect the paths of quotations. Rule 20 removes path information that no longer holds. Note the condition that $V \neq X$, which keeps V and X from being matched into the same entity — certificates distributed by the signer are based on the signer's knowledge, not a quotation. If all support is missing after the application of rule 20, rule 21 detects the lack of support and removes the unsupported quotation.

The second way that revocation can propagate in the graph is by removing deduced knowledge. Rule 22 updates the support set of deduced



Rule 22: Updating the support set



Rule 23: Updating the support set

knowledge when supporting certificates are removed, to keep the path information current. Note the condition that $\omega \neq \epsilon$ (where ϵ is the empty string), which prevents the support set of type O public-key knowledge of being removed.

Rules 23 works in the same way as 22, but with the assumption that Z's knowledge is inactive.

5.6.4 Example

C decides to remove the quotation previously given to D, and applies rule 19 to remove the certificate. The result is shown in figure 5.9.

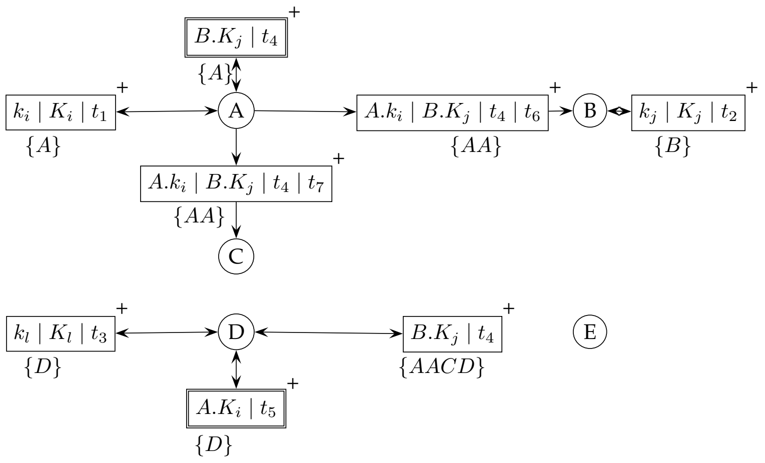


Figure 5.9. C removes the quotation made to D

Now there is no longer support for D's deduced knowledge about $B.K_j$, so the deductive rules come into play. Rule 18 detects the missing quotation to D, and removes the deduction. The result is shown in figure 5.10.

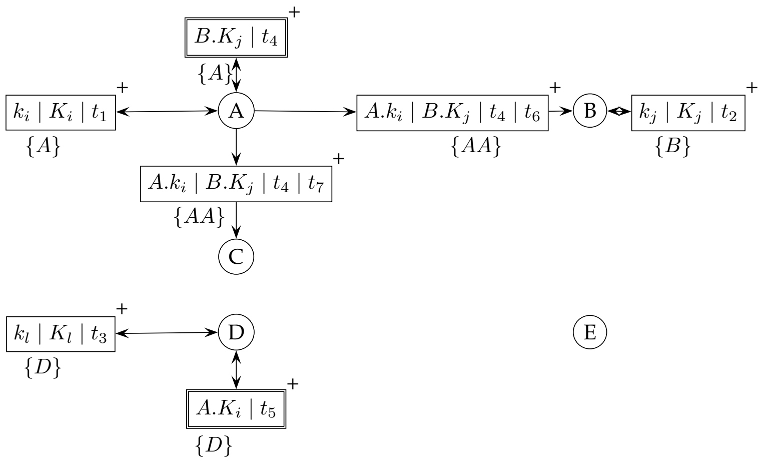


Figure 5.10. D's unsupported type I public-key knowledge is removed

5.7 Revocation by Inactivation

To capture the mechanics of revocation due to expiration, we use inactivation. By inactivating knowledge, entities can hold and spread information on the form "$X.K_i$ was X's public key, but since time t_e it is not valid". This may be necessary not only when a key has reached its expiration time, but also when a key simply should not be used any longer for non-malicious reasons, e.g. when a user has changed work assignments. Therefore, we call this procedure inactivation, and note that expiration is only one of several reasons why it should be performed.

Users can inactivate their own keys, as well as those of others. Even when a key is inactive it is necessary to store it, to be able to verify signatures that were made while the key was still valid. However, signatures that have been created after its inactivation should not be considered valid. We address this issue by keeping inactivated certificates in the graph, but not letting entities accept signatures made with keys that they know are inactive (as ensured by rule 11). Furthermore, once an entity has inactivated their own key, or knows that their key has been inactivated by another entity, they are no longer able to sign certificates with that key (as ensured by rules 4 and 6), and any signatures made between the time of inactivation and the time that the key owner learns of the inactivation will be removed (as ensured by rule 14).

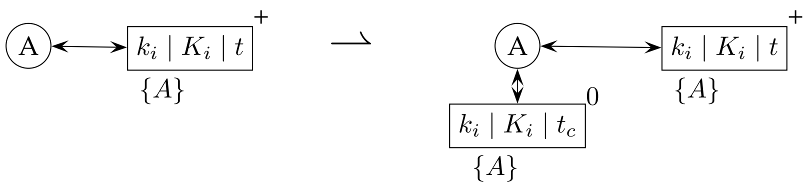
Inactive knowledge and certificates are marked with a zero in the upper right corner. When an inactive collision occurs, the policy will be to give precedence to inactive knowledge and certificates over positive ones. When an inactive knowledge or certificate is added where there is already a corresponding positive statement, the inactive will be spread along any existing positive paths. Following this, the positive statements will be removed from the graph (or, in some cases, inactivated).

More formally, precedence is given to inactive statements through the following policy:

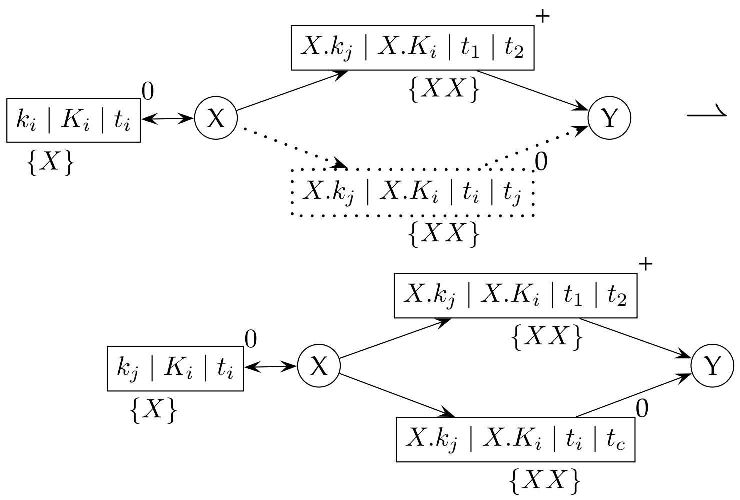
(1) When a user has an inactive collision caused by the inactive statement ρ, follow the paths of positive certificates on the same form as ρ that emanate from that user, and add inactive certificates along those paths.

(2) When the end of a path is reached, backtrack along the path and remove the positive certificates on the same form as ρ that are distributed to entities on the path.

5.7.1 Inactivation of Keypair Knowledge



Rule 24: Inactivating keypair knowledge

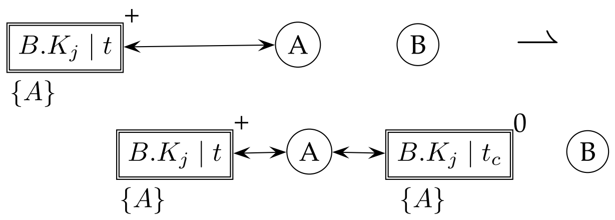

$X.k_j \mid X.K_i \mid t_1 \mid t_2$
+
$\{XX\}$
$k_i \mid K_i \mid t_i$
0
$\{X\}$
X
Y
$X.k_j \mid X.K_i \mid t_i \mid t_j$
0
$\{XX\}$
$k_j \mid K_i \mid t_i$
$X.k_j \mid X.K_i \mid t_i \mid t_c$

Rule 25: Propagating inactive keypair knowledge along positive paths

Entities may inactivate their own keys by using the interactive rule 24. The inactivated keypair knowledge will be stamped with the current time t_c. Applying this rule can affect the graph through the deductive rule 25, which describes how certificates for the inactivated key are propagated along the paths of positive certificates for the key. The reason for this propagation is to spread information about the inactivation to any user that previously had information about the positive certification. While rule 25 initiates the propagation, its continuation is ensured by rule 30, which handles propagation along general certificate paths. Note that the positive knowledge is kept by these rules — the reason for this is that any positive certificates issued by A should be kept until the propagation is complete (if the positive knowledge was not kept, rules 15, 20 and 21 would remove the positive certificates). After the propagation, rule 33 will remove the positive knowledge.

Once an entity has inactivated one of its keypairs, it cannot be used to sign new certificates, as ensured by rules 4 and 6. Rule 6 lets users create certificates for inactive keypair knowledge, as long as the signing key is not inactive.

5.7.2 Inactivation of Public-Key Knowledge



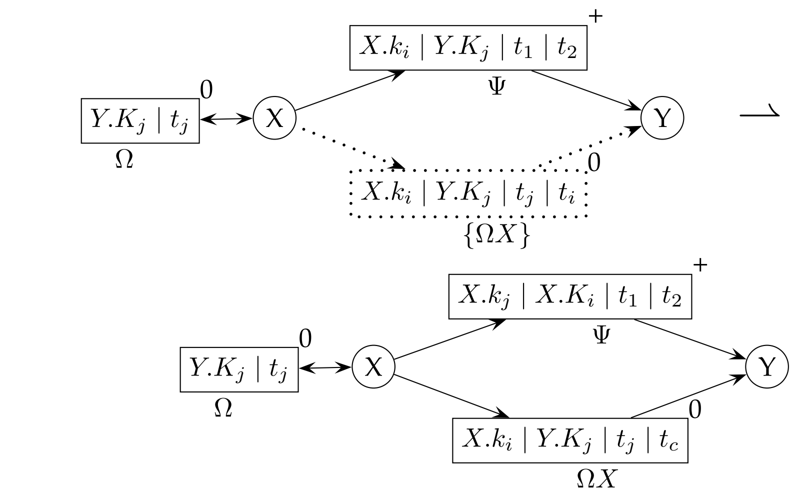
Rule 26: Inactivating public-key knowledge

Keys can also be inactivated by other entities than the key owner. Type O public-key knowledge can be inactivated interactively via rule 26. The deductive rule 27 will initiate the propagation of the inactivation via positive paths, and this propagation will be continued by rule 30. The positive knowledge is kept for the same reason as in rule 24, and will be removed after propagation by rule 34.

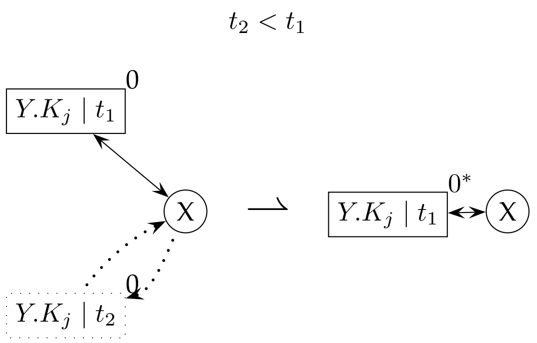
The procedures for creating and updating certificates for inactive public-key knowledge are described by rules 4 and 5.

When a key is inactivated by some other entity than the key owner, the key owner may not be aware of the inactivation for some time, and may therefore use the key. When the key owner does deduce knowledge about the inactivated key, all certificates signed with that key since the time of inactivation will be removed by the deductive rule 14.

Different entities may inactivate a key at different times. Since an entity may receive certificates for the inactivation of a given key from different sources, they may end up with deduced public-key knowledge of that key

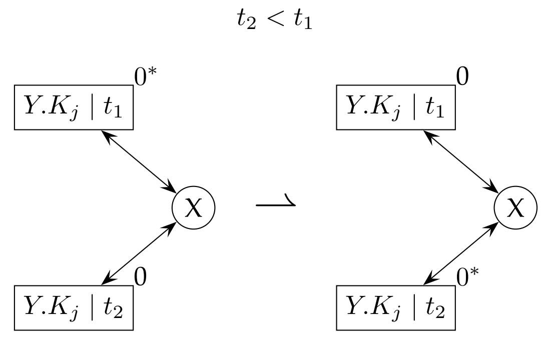

$Y.K_j \mid t_j$
0
Ω
X
$X.k_i \mid Y.K_j \mid t_1 \mid t_2$
+
Ψ
Y
$X.k_i \mid Y.K_j \mid t_j \mid t_i$
0
$\{\Omega X\}$
$\rightharpoondown$
$Y.K_j \mid t_j$
0
Ω
X
$X.k_j \mid X.K_i \mid t_1 \mid t_2$
+
Ψ
Y
$X.k_i \mid Y.K_j \mid t_j \mid t_c$
0
ΩX

Rule 27: Propagating inactive public-key knowledge along positive paths

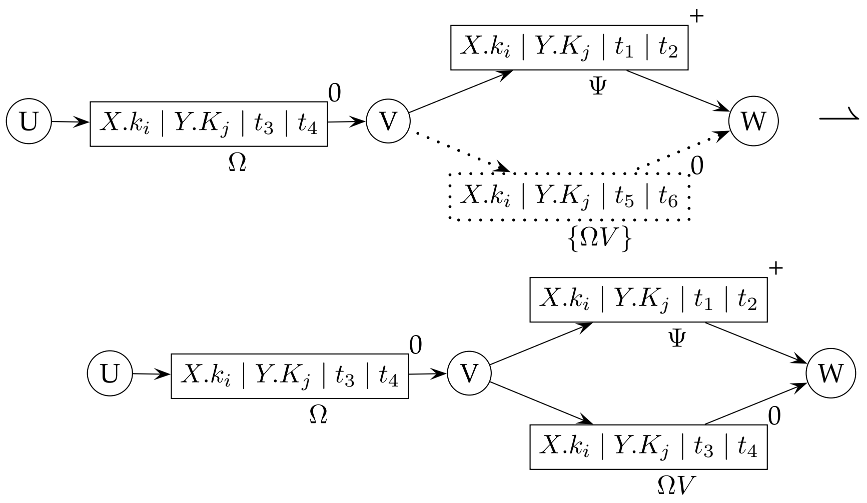


Rule 28: Finding the earliest inactivation time

with different time stamps. In this case, it is important that the entity considers the earliest possible inactivation they have access to — failing to do so means possibly accepting signatures that were made after the key was inactivated. To keep track of the earliest inactivation for a specific public key, the deductive rules 28 and 29 mark it with an asterisk.



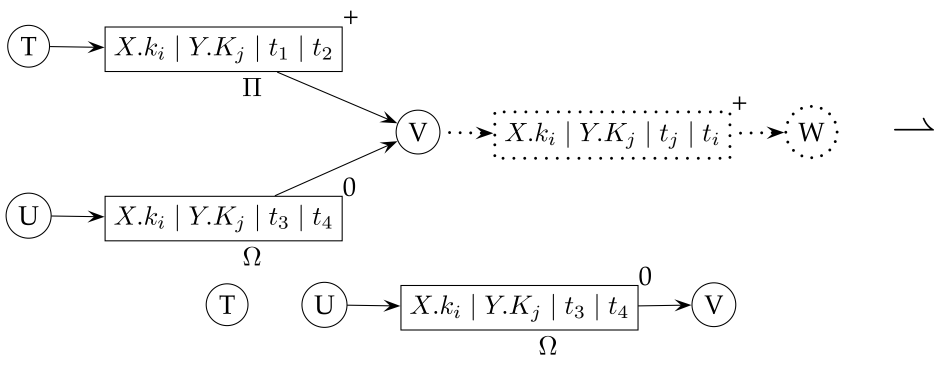
Rule 29: Updating the earliest inactivation time


U
$X.k_i \mid Y.K_j \mid t_3 \mid t_4$
0
Ω
V
$X.k_i \mid Y.K_j \mid t_1 \mid t_2$
+
Ψ
W
$X.k_i \mid Y.K_j \mid t_5 \mid t_6$
0
$\{\Omega V\}$
U
$X.k_i \mid Y.K_j \mid t_3 \mid t_4$
0
Ω
V
$X.k_i \mid Y.K_j \mid t_1 \mid t_2$
+
Ψ
W
$X.k_i \mid Y.K_j \mid t_3 \mid t_4$
0
ΩV

Rule 30: Propagating inactive certificates along positive paths

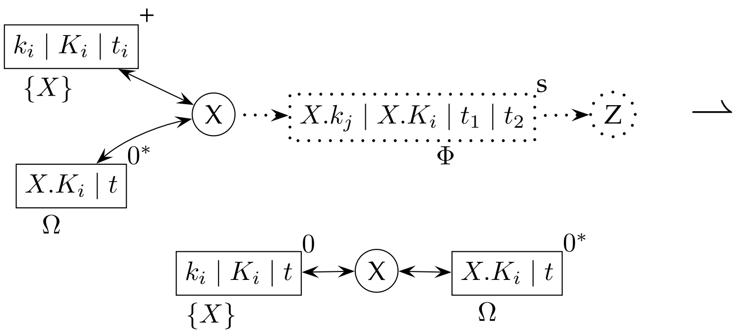
5.7.3 Inactive Certificates

Certificates themselves cannot be interactively inactivated — the inactivation of a certificate must be based on inactive keypair or public-key knowledge. However, there are deductive rules that inactivate certificates when necessary. As already mentioned, rules 25 and 27 initiate the propagation of inactive knowledge along paths of positive certificates. Rules 30 and 31 continue and end the process. When an inactive certificate σ is added to



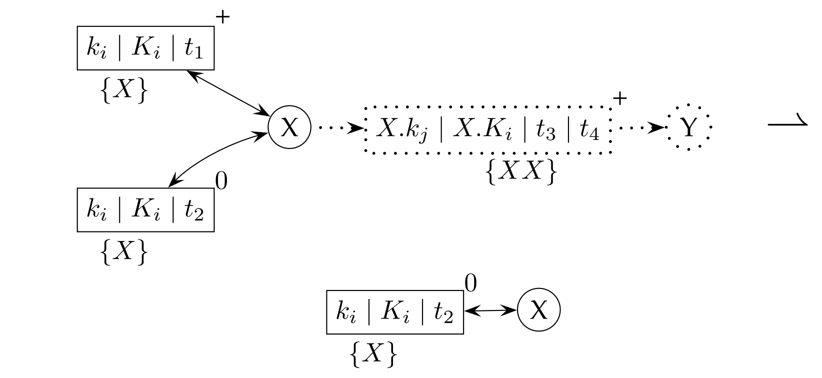
T
$X.k_i \mid Y.K_j \mid t_1 \mid t_2$
+
Π
V
$X.k_i \mid Y.K_j \mid t_j \mid t_i$
+
W
U
$X.k_i \mid Y.K_j \mid t_3 \mid t_4$
0
Ω
T
U
$X.k_i \mid Y.K_j \mid t_3 \mid t_4$
0
Ω
V

Rule 31: Backtracking along positive paths

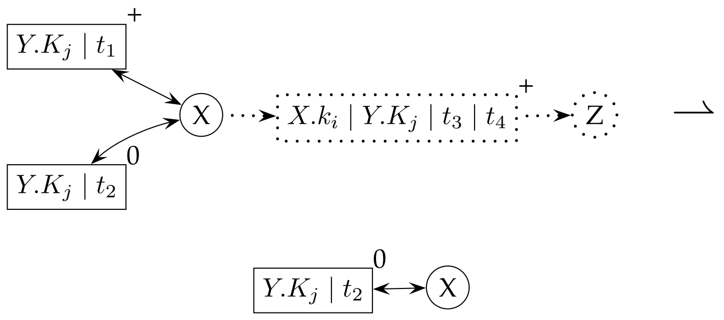


Rule 32: Inactivating positive keypair knowledge

a C-graph, rule 30 will propagate it along the paths of the positive certificates on the same form as σ. Rule 31 detects the end of each such path and removes the last positive certificate on the path, thus back-tracking to the beginning of quoted paths until no positive certificates for the same binding as σ are left. Rules 33 and 34 remove the positive keypair or public-key knowledge at the root of the paths when propagation and backtracking are complete. Rules 35, 36, 37, and 38 remove incoming positive certificates when they appear as part of the support for a deduction about the key. This is done in order to ensure that new positive knowledge cannot be deduced about the key, and to remove deductions that are already in place.



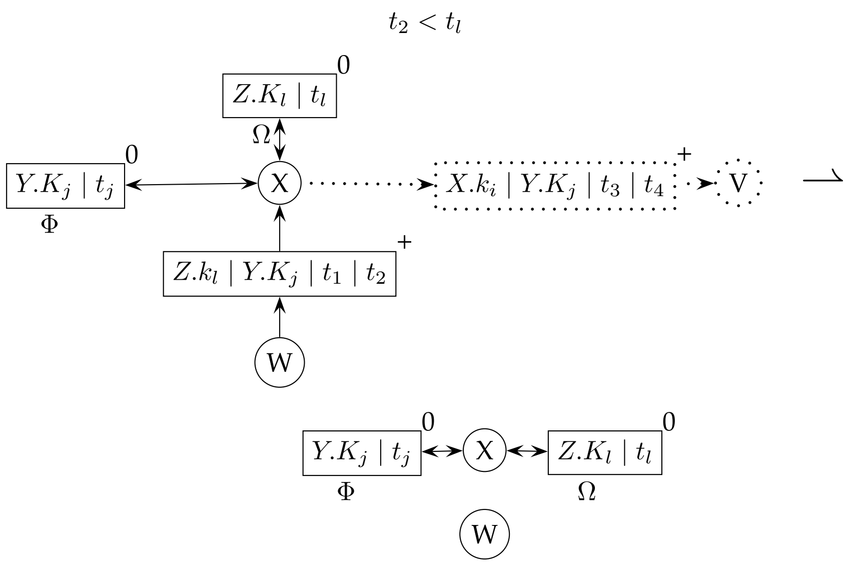
Rule 33: Removing positive keypair knowledge



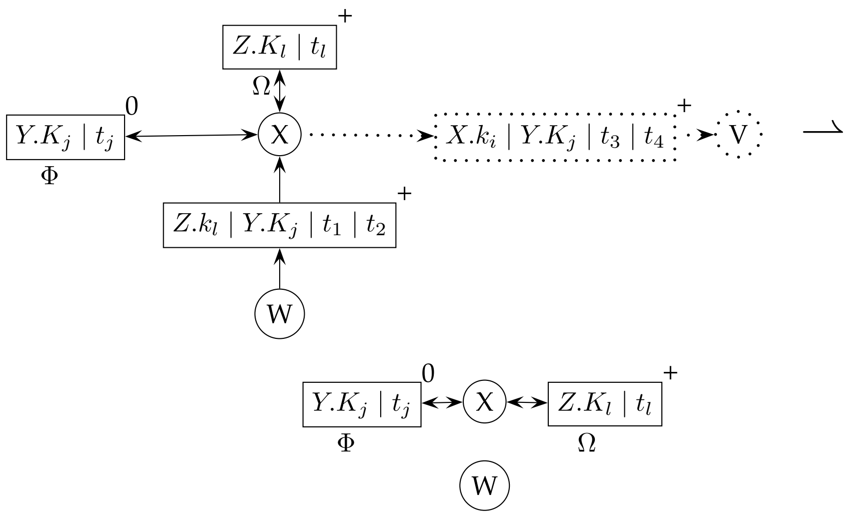
Rule 34: Removing positive public-key knowledge

5.7.4 Removal of Inactive Statements

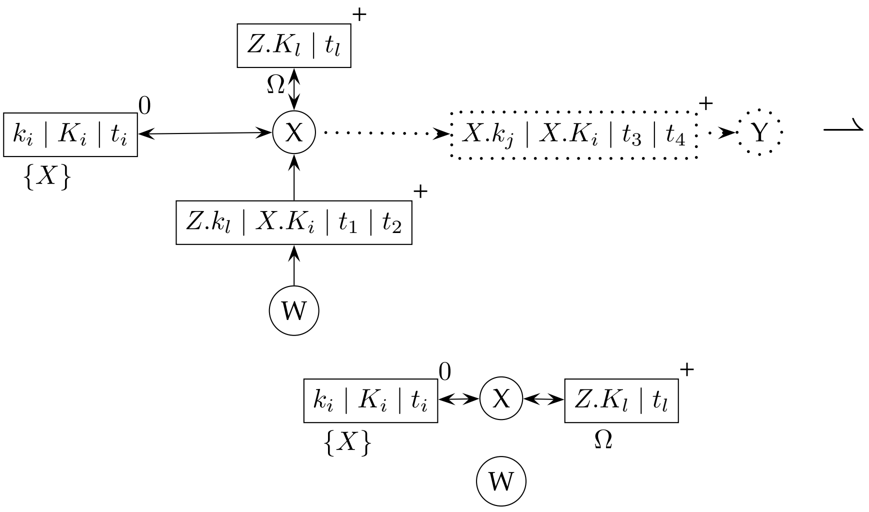
Inactive knowledge and certificates can be removed in a similar way to their positive counterparts. However, we do not need revocation rules specific to inactive statements, since the intended results are achieved by the rules 13–23. Removal of an inactive certificate can be necessary when deduced knowledge loses its support, or when information about the inactive key is not needed in the system any longer.



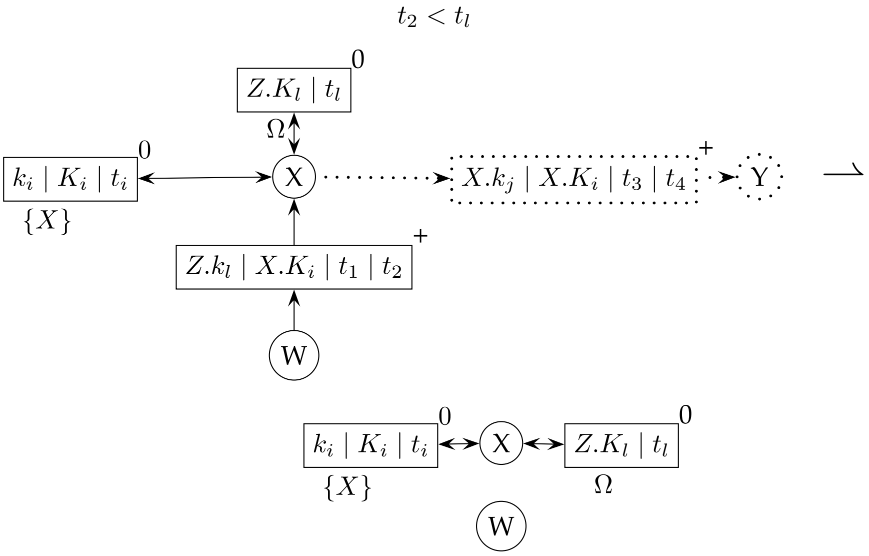
Rule 35: Removing the support for positive public-key knowledge



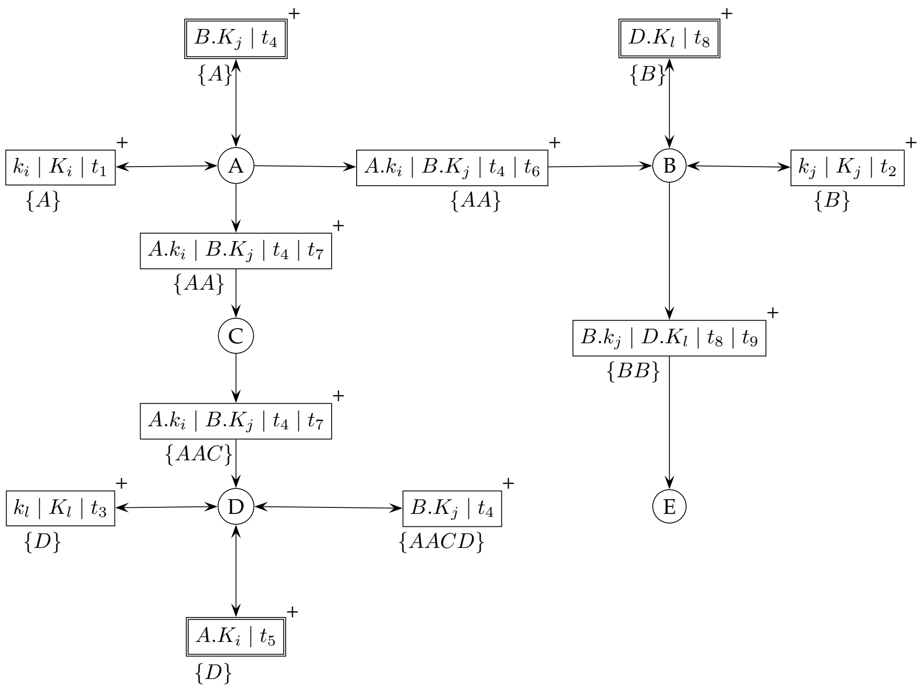
Rule 36: Removing the support for positive public-key knowledge



Rule 37: Removing the support for positive public-key knowledge



Rule 38: Removing the support for positive public-key knowledge



$B.K_j \mid t_4$ + $\{A\}$
$D.K_l \mid t_8$ + $\{B\}$
$k_i \mid K_i \mid t_1$ + $\{A\}$
A
$A.k_i \mid B.K_j \mid t_4 \mid t_6$ + $\{AA\}$
B
$k_j \mid K_j \mid t_2$ + $\{B\}$
$A.k_i \mid B.K_j \mid t_4 \mid t_7$ + $\{AA\}$
C
$B.k_j \mid D.K_l \mid t_8 \mid t_9$ + $\{BB\}$
$A.k_i \mid B.K_j \mid t_4 \mid t_7$ + $\{AAC\}$
$k_l \mid K_l \mid t_3$ + $\{D\}$
D
$B.K_j \mid t_4$ + $\{AACD\}$
E
$A.K_i \mid t_5$ + $\{D\}$

Figure 5.11. B adds knowledge and certificate about $D.K_l$

5.7.5 Example

Assume a system graph that looks like in figure 5.8, and that B adds outside knowledge about $D.K_l$ in a certificate that they also quote to E. This will give us the system graph in figure 5.11.

Now assume that A decides to inactivate their public-key knowledge of $B.K_j$, and applies rule 26 to do so — this will add a new inactive knowledge with a current time stamp. Rule 28 will mark the knowledge with an asterisk, since it is the earliest inactivation of $B.K_j$ in A's possession. Rule 27 will now detect that A holds inactive knowledge and distributes positive certificates for the same key, and add inactive certificates between A and B, and between A and C. Again, rule 27 applies and adds the inactive certificate between C and D. This will produce the graph shown in figure 5.12.

At this point, the ends of the paths of quoted positive certificates have been reached at B and at D. Rule 31 detects that the ends are reached, and backtracks to remove the three positive certificates along the paths. At D, there is no longer an incoming certificate for $B.K_j$, so rule 18 removes

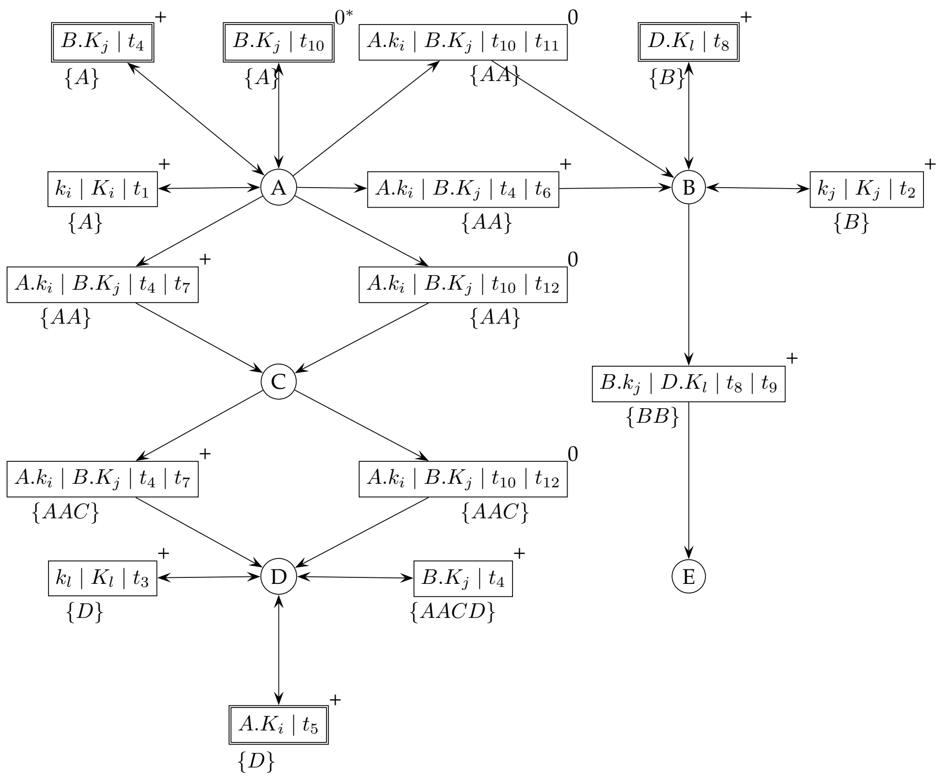
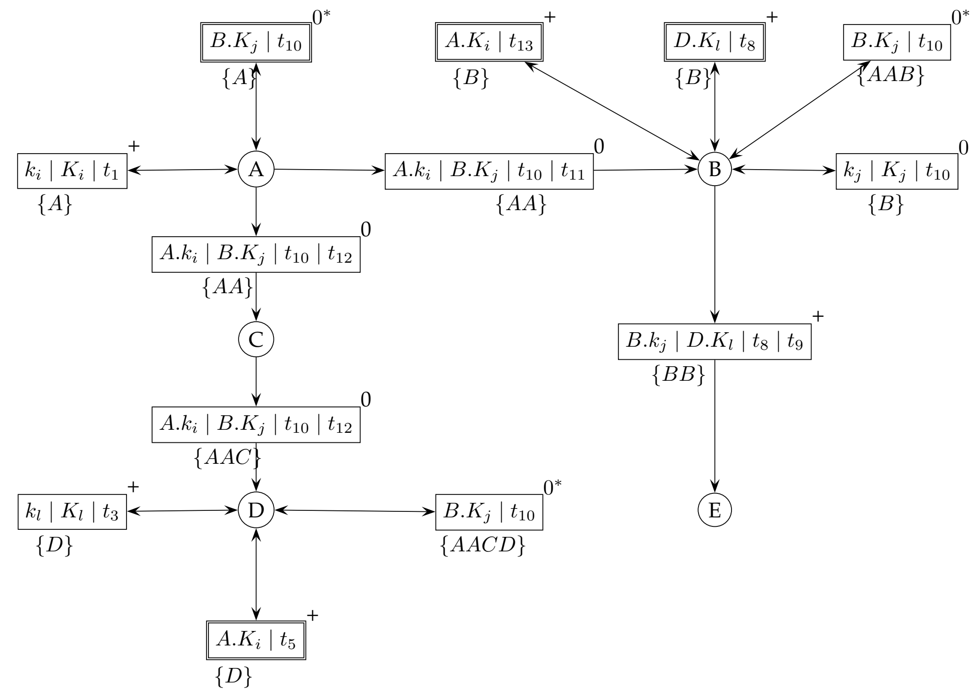


Figure 5.12. A's inactivation is spread in the graph

D's deduced knowledge. Rule 9 deduces new knowledge for D about the inactivation of $B.K_j$, and rule 28 marks it with an asterisk since it is the earliest inactivation for $B.K_j$ in D's possession. At A, rule 34 removes the positive public-key knowledge for $B.K_j$.

Now assume that B adds outside public-key knowledge of $A.K_j$, using rule 3. The presence of the new knowledge triggers rule 9, which deduces knowledge of the inactivation of $B.K_j$ at B. Rule 28 marks it with an asterisk. Rule 31 detects B's positive keypair knowledge and inactive public-key knowledge, and inactivates the keypair knowledge. Since B signed the certificate for $D.K_l$ before t_{10}, there is still support for the certificate and it will not be removed. The result is shown in figure 5.13.



$B.K_j \mid t_{10}$ 0^* $\{A\}$
$A.K_i \mid t_{13}$ $+$ $\{B\}$
$D.K_l \mid t_8$ $+$ $\{B\}$
$B.K_j \mid t_{10}$ 0^* $\{AAB\}$
$k_i \mid K_i \mid t_1$ $+$ $\{A\}$
A
$A.k_i \mid B.K_j \mid t_{10} \mid t_{11}$ 0 $\{AA\}$
B
$k_j \mid K_j \mid t_{10}$ 0 $\{B\}$
$A.k_i \mid B.K_j \mid t_{10} \mid t_{12}$ 0 $\{AA\}$
C
$B.k_j \mid D.K_l \mid t_8 \mid t_9$ $+$ $\{BB\}$
$A.k_i \mid B.K_j \mid t_{10} \mid t_{12}$ 0 $\{AAC\}$
$k_l \mid K_l \mid t_3$ $+$ $\{D\}$
D
$B.K_j \mid t_{10}$ 0^* $\{AACD\}$
E
$A.K_i \mid t_5$ $+$ $\{D\}$

Figure 5.13. Positive certificates are removed along the paths

5.8 Revocation by Negation

There are many times when a revocation should be interpreted as the revoking user making a negative assertion about the authenticity of a binding, e.g. when the certified key or the signing key has been compromised, and entities must be prevented from believing in a certain binding. Negating a statement is the strongest way to revoke; by doing so, entities can spread information on the form "$X.K_i$ is *not* X's public key".

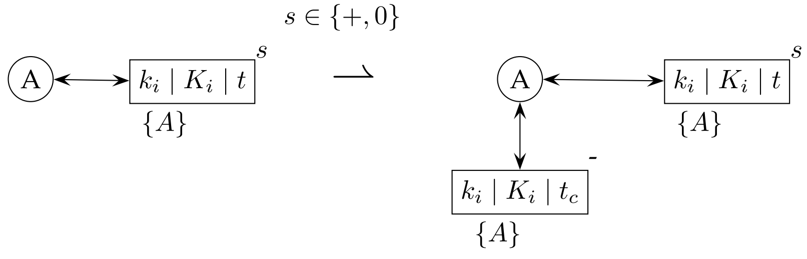
Negative knowledge and certificates are marked with a minus sign in the upper right corner. When a negative collision occurs, the policy will be to give precedence to negative statements. When a negative statement is added where there is already a corresponding positive or inactive statement, the negative will be spread along any existing positive or inactive paths, and the positive and inactive statements will then be removed from the graph (or, in some cases, negated). Negative certificates cannot be inactivated, but may be removed and revoked if they lose their support.

We have chosen to give precedence to negative statements through the following policy:

(1) When a user has a negative collision, follow the paths of positive and/or inactive certificates that emanate from that user, and add negative certificates along those paths.

(2) When the end of a path is reached, backtrack along the path and remove the corresponding positive and/or inactive certificates distributed to entities along the path.

5.8.1 Negation of Keypair Knowledge



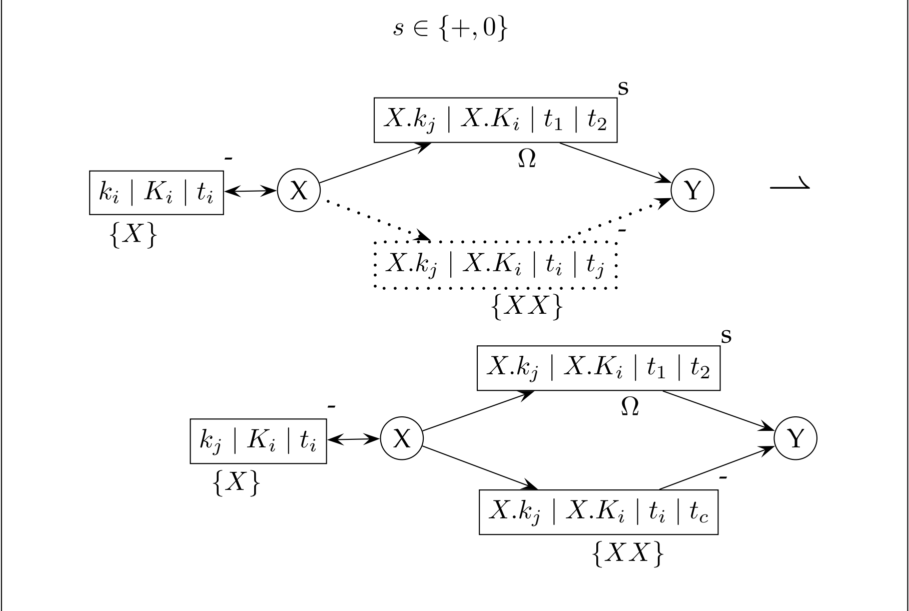
Rule 39: Negating keypair knowledge

Positive or inactive keypair knowledge can be interactively negated by rule 39 — the knowledge will be time-stamped with the current time t_c. The existing knowledge is kept in the graph until propagation is finished. The deductive rule 40 spreads the negation by distributing negated certificates to recipients of positive certificates for the same binding, in the same way as for inactivation. The propagation is continued by rule 43, and the backtracking process is finished by rule 46.

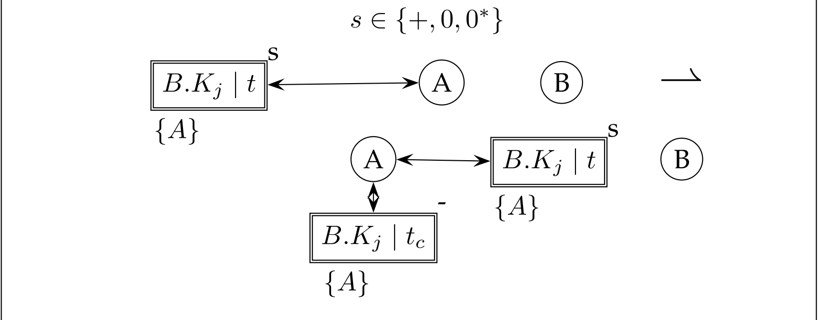
Certificates for negative keypair knowledge can be created by rule 6.

5.8.2 Negation of Public-Key Knowledge

Public-key knowledge that is positive or inactive can be interactively negated via rule 41. The existing knowledge is kept until propagation is finished in the same way as for inactivation. Rule 42 initiates the propagation of the negation on positive and inactive paths; the propagation is continued by rule 43 and the backtracking is finished by rule 47.


$s \in \{+, 0\}$
$X.k_j \mid X.K_i \mid t_1 \mid t_2$
s
Ω
-
$k_i \mid K_i \mid t_i$
$\{X\}$
X
Y
$X.k_j \mid X.K_i \mid t_i \mid t_j$
$\{XX\}$
$k_j \mid K_i \mid t_i$
$X.k_j \mid X.K_i \mid t_i \mid t_c$

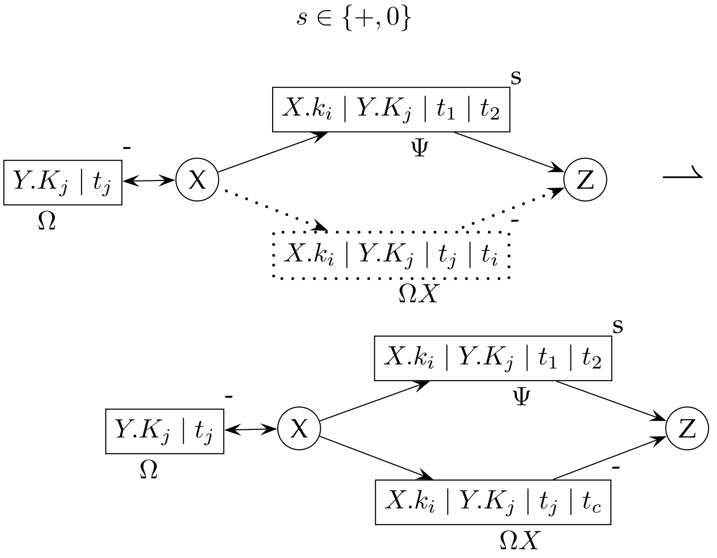
Rule 40: Propagating negative keypair knowledge



Rule 41: Negating public-key knowledge

When an entity learns that one of their keys has been negated, all certificates signed with that key are removed. This is achieved by rule 45, which negates the knowledge (after any necessary propagation on paths of certificates for that key), and rule 15, which removes certificates without support.

Certificates for negative public-key knowledge can be created by rule 4.



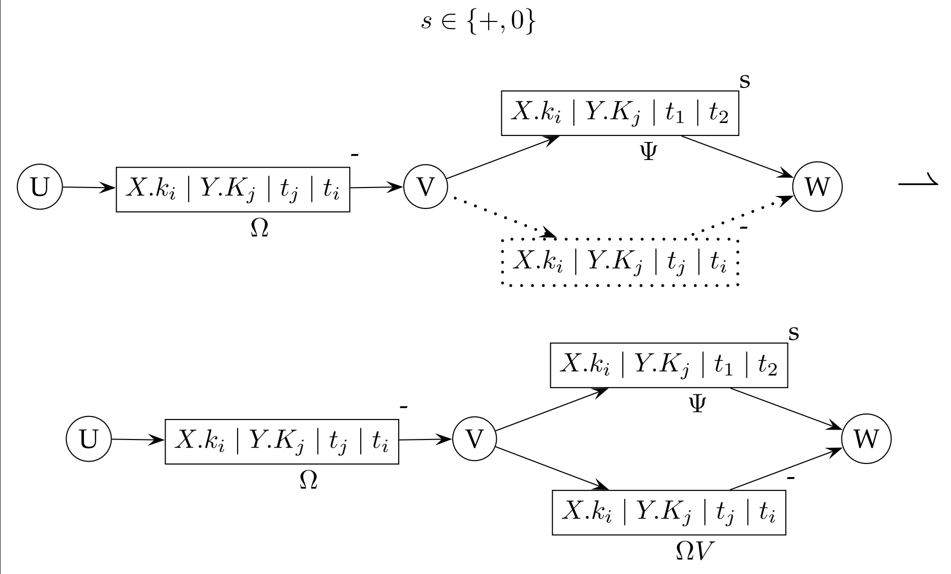
$s \in \{+, 0\}$
s
$X.k_i \mid Y.K_j \mid t_1 \mid t_2$
Ψ
-
$Y.K_j \mid t_j$
Ω
X
Z
-
$X.k_i \mid Y.K_j \mid t_j \mid t_i$
ΩX
$X.k_i \mid Y.K_j \mid t_j \mid t_c$

Rule 42: Propagating negative public-key knowledge

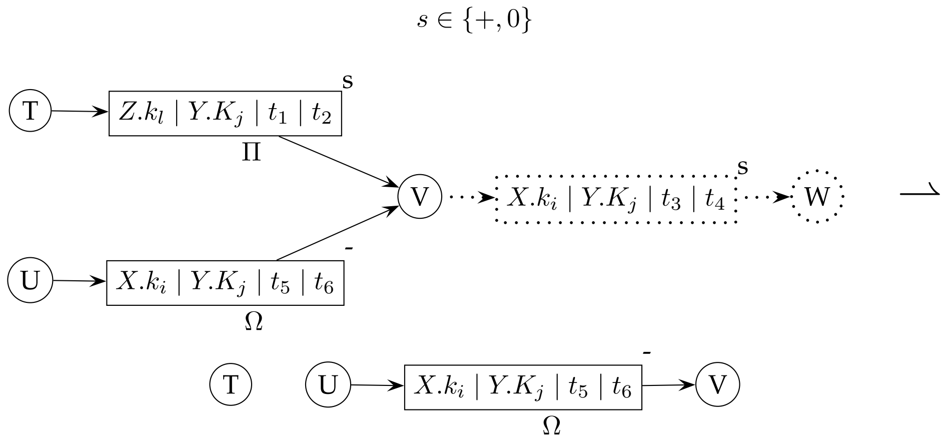
5.8.3 Negation of Certificates

Like inactivation, negation cannot be applied directly to a certificate — a certificate must be based on keypair or public-key knowledge. However, once knowledge about a key has been negated, certificates on paths stemming from that knowledge are negated as well.

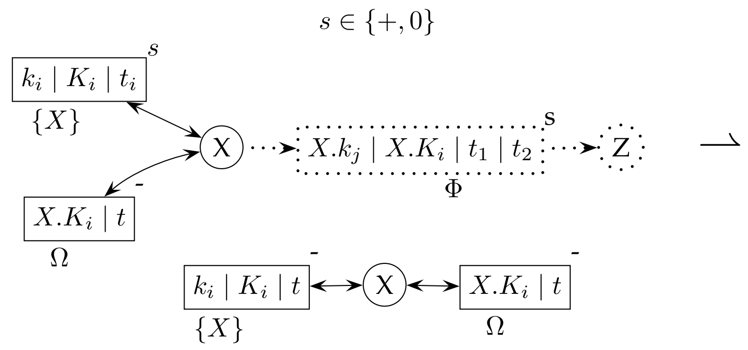
Rules 43 and 44 work together in the same way as rules 30 and 31. When a negative certificate ρ is added to a C-graph by either of the rules 40 or 42, rule 43 will propagate it along the paths of the positive or inactive certificates on the same form as ρ. Rule 44 detects the end of each such path and removes the last positive or inactive certificate on the path, thus back-tracking to the beginning until no positive or inactive certificates on the same for as ρ are left. Rules 46 and 47 remove the positive or inactive knowledge at the beginning of the path when the propagation of negative statements and backtracking are finished. Rules 48, 49, 50, and 51 remove incoming positive or inactive certificates for the negated key, when they form part of the support for a deduction about the negated key. This is done in order to make sure that no new positive or inactive knowledge will be deduced about the key — removing a deduced knowledge is not enough; if there is support for it, it will automatically reappear.


$s \in \{+, 0\}$
U
$X.k_i \mid Y.K_j \mid t_j \mid t_i$
-
Ω
V
$X.k_i \mid Y.K_j \mid t_1 \mid t_2$
s
Ψ
W
$X.k_i \mid Y.K_j \mid t_j \mid t_i$
-
ΩV

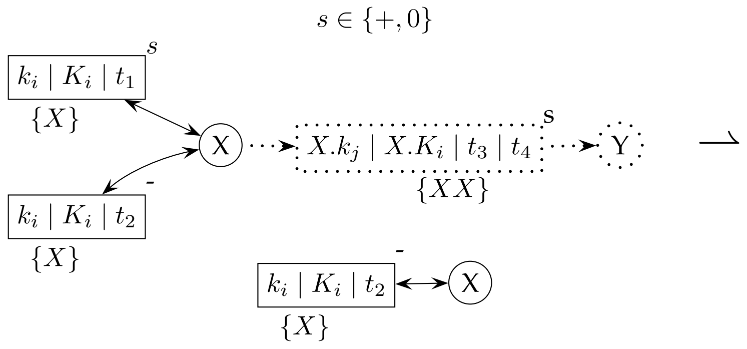
Rule 43: Propagating negative certificates


$s \in \{+, 0\}$
T
$Z.k_l \mid Y.K_j \mid t_1 \mid t_2$
s
Π
V
$X.k_i \mid Y.K_j \mid t_3 \mid t_4$
W
U
$X.k_i \mid Y.K_j \mid t_5 \mid t_6$
-
Ω

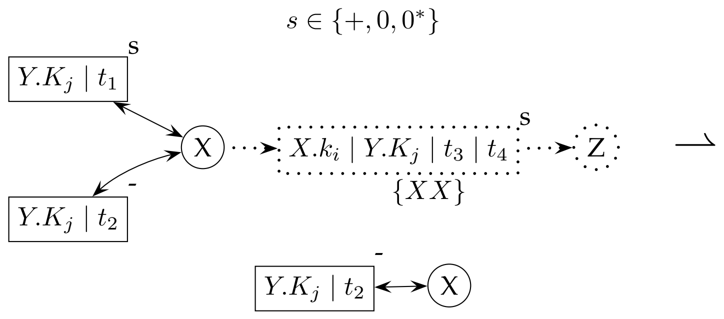
Rule 44: Backtracking



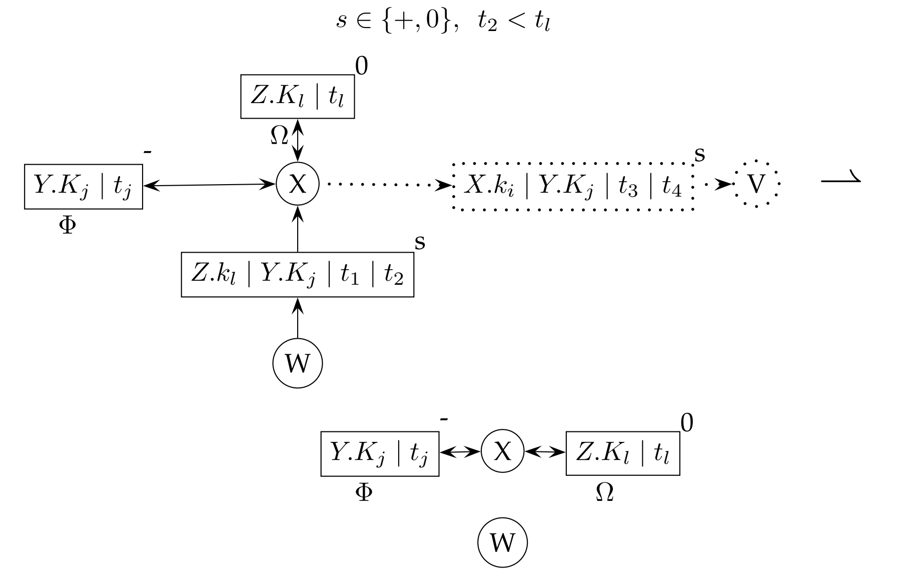
Rule 45: Negating positive/inactive keypair knowledge



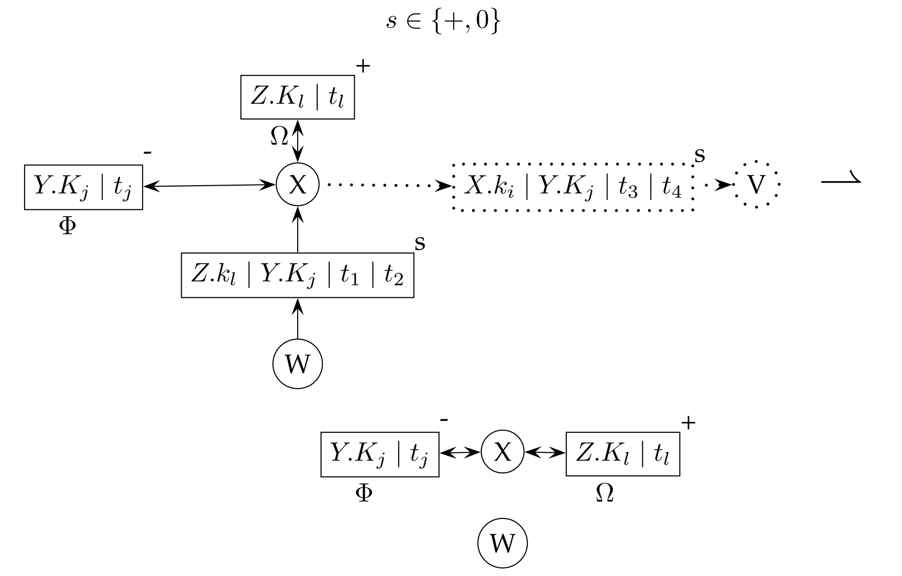
Rule 46: Removing positive/inactive keypair knowledge



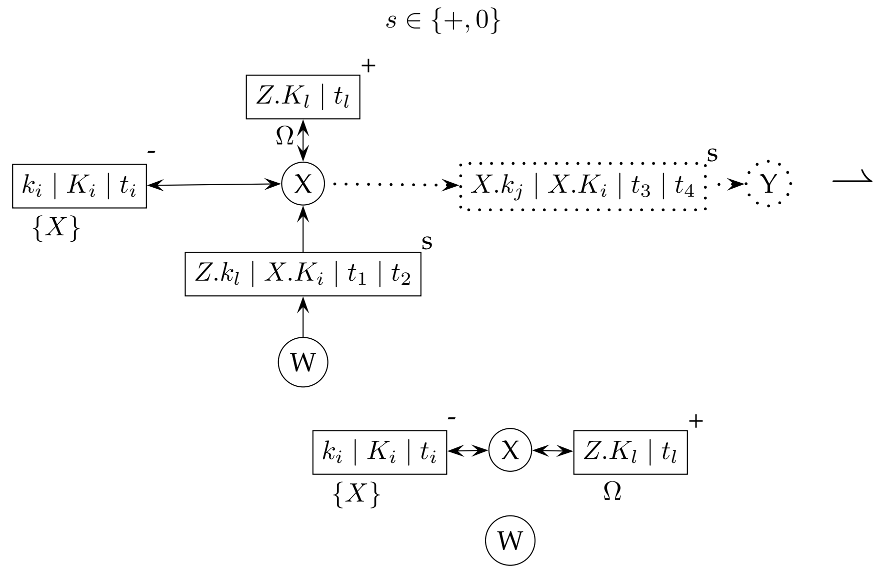
Rule 47: Removing positive/inactive public-key knowledge



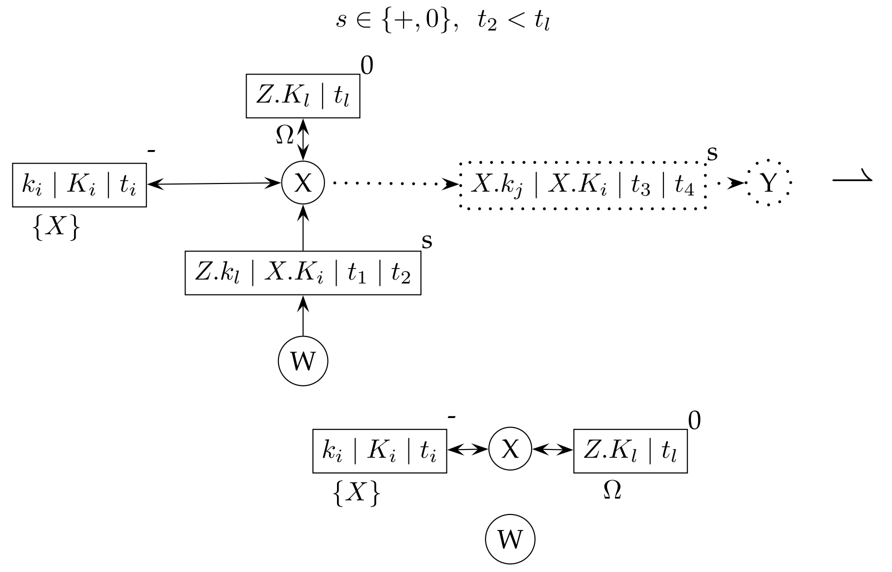
Rule 48: Removing the support for positive/inactive public-key knowledge



Rule 49: Removing the support for positive/inactive public-key knowledge



Rule 50: Removing the support for positive/inactive public-key knowledge


$s \in \{+, 0\}, \;\; t_2 < t_l$
$Z.K_l \mid t_l$
0
Ω
-
$k_i \mid K_i \mid t_i$
$\{X\}$
X
$X.k_j \mid X.K_i \mid t_3 \mid t_4$
s
Y
$Z.k_l \mid X.K_i \mid t_1 \mid t_2$
s
W
$\rightharpoondown$

Rule 51: Removing the support for positive/inactive public-key knowledge

5.8.4 Removal of Negative Statements

The revocation rules 13–23 apply for negative knowledge and certificates as well, giving a functionality for revocation of negative statements. The main reason for revocation of a negative statement is that a deduction loses its support — possibly because the knowledge of the signing key needed in the deduction has been revoked — but it is also possible that a negative certificate has been issued by mistake and should simply be removed, along with any knowledge based on that certificate.

5.8.5 Example

Following the situation in figure 5.13, B negates their knowledge of $D.K_l$, and D negates their knowledge of $A.K_i$, both using rule 41. This gives the graph shown in figure 5.14.

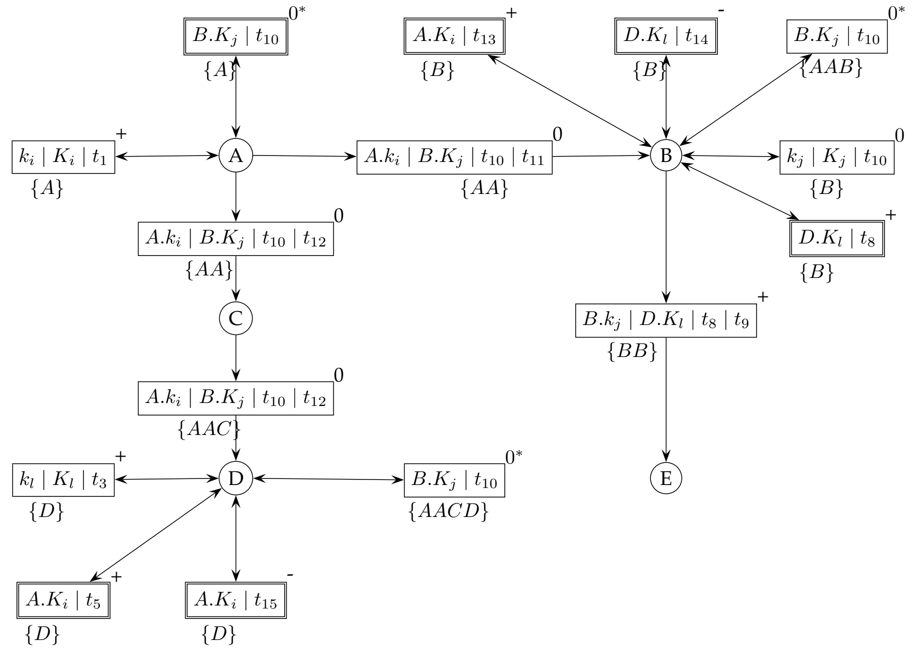

$B.K_j \mid t_{10}$ 0* $\{A\}$
$A.K_i \mid t_{13}$ + $\{B\}$
$D.K_l \mid t_{14}$ - $\{B\}$
$B.K_j \mid t_{10}$ 0* $\{AAB\}$
$k_i \mid K_i \mid t_1$ + $\{A\}$
A
$A.k_i \mid B.K_j \mid t_{10} \mid t_{11}$ 0 $\{AA\}$
B
$k_j \mid K_j \mid t_{10}$ 0 $\{B\}$
$A.k_i \mid B.K_j \mid t_{10} \mid t_{12}$ 0 $\{AA\}$
$D.K_l \mid t_8$ + $\{B\}$
C
$B.k_j \mid D.K_l \mid t_8 \mid t_9$ + $\{BB\}$
$A.k_i \mid B.K_j \mid t_{10} \mid t_{12}$ 0 $\{AAC\}$
$k_l \mid K_l \mid t_3$ + $\{D\}$
D
$B.K_j \mid t_{10}$ 0* $\{AACD\}$
E
$A.K_i \mid t_5$ + $\{D\}$
$A.K_i \mid t_{15}$ - $\{D\}$

Figure 5.14. B and D negate their knowledge

As regards B, rule 42 detects the combination of negative knowledge and the positive certificate distributed to E, and adds a negative certificate to E as well. Rule 44 then detects that E has both a positive and a negative certificate for $D.K_l$, and no outgoing certificates, and removes the positive certificate issued by B. Next, rule 47 detects B's combination of positive and negative knowledge for $D.k_l$, and removes the positive.

As for D, rule 47 detects the combination of positive and negative knowledge for $A.K_i$, and removes the positive. Rule 18 detects the lack of support for the deduction of $B.K_j$, and removes it.

The resulting graph is shown in figure 5.15.

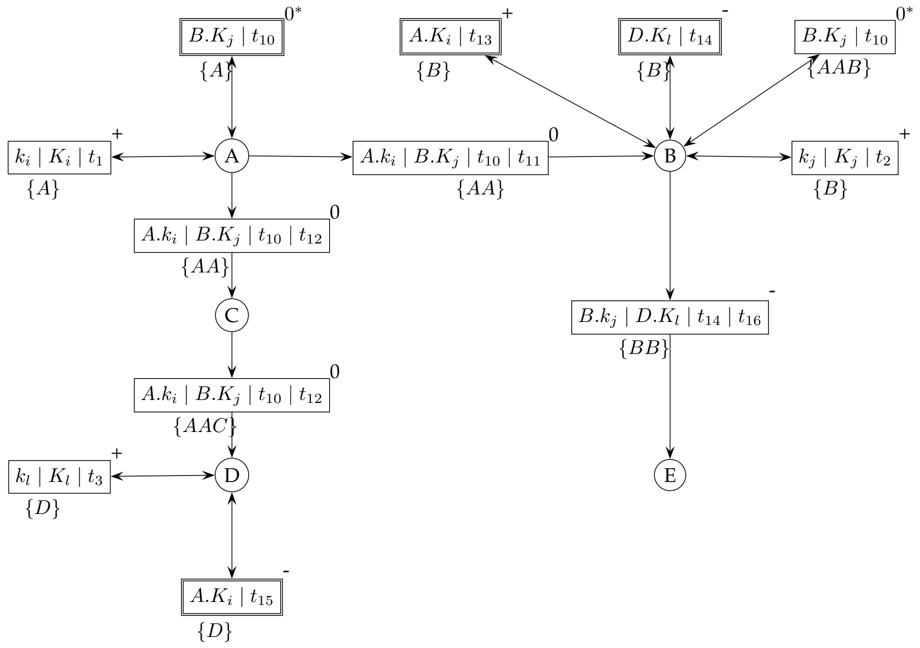

$B.K_j \mid t_{10}$ 0*
$\{A\}$
$A.K_i \mid t_{13}$ +
$\{B\}$
$D.K_l \mid t_{14}$ -
$\{B\}$
$B.K_j \mid t_{10}$ 0*
$\{AAB\}$
$k_i \mid K_i \mid t_1$ +
$\{A\}$
A
$A.k_i \mid B.K_j \mid t_{10} \mid t_{11}$ 0
$\{AA\}$
B
$k_j \mid K_j \mid t_2$ +
$\{B\}$
$A.k_i \mid B.K_j \mid t_{10} \mid t_{12}$ 0
$\{AA\}$
C
$B.k_j \mid D.K_l \mid t_{14} \mid t_{16}$ -
$\{BB\}$
$A.k_i \mid B.K_j \mid t_{10} \mid t_{12}$ 0
$\{AAC\}$
$k_l \mid K_l \mid t_3$ +
$\{D\}$
D
E
$A.K_i \mid t_{15}$ -
$\{D\}$

Figure 5.15. The negations are propagated

5.9 Summary: Interactive Rules

The interactive rules available for entities to create, quote, and revoke knowledge and certificates are summarized in table 5.1. All other rules for the C-graph are deductive, except for rule 1, which is administrative.

		Creation	Quotation	Removal	Inactivation	Negation
Keypair knowledge	+	Rule 2	NA[b]	Rule 13	Rule 24	Rule 39
	0	NA[a]	NA[b]	Rule 13	NA[c]	Rule 39
	–	NA[a]	NA[b]	Rule 13	NA[d]	NA[e]
Public-key knowledge, Type O	+	Rule 3	NA[b]	Rule 16	Rule 26	Rule 41
	0	NA[a]	NA[b]	Rule 16	NA[c]	Rule 41
	–	NA[a]	NA[b]	Rule 16	NA[d]	NA[e]
Certificates	+	Rules 4,6	Rule 7	Rule 19	NA[f]	NA[g]
	0	Rules 4,6	Rule 7	Rule 19	NA[f]	NA[g]
	–	Rules 4,6	Rule 7	Rule 19	NA[f]	NA[g]

[a] Only positive knowledge can be created
[b] Creating a certificate can be viewed as quotation of knowledge
[c] Inactive knowledge cannot be inactivated
[d] Negative knowledge cannot be inactivated
[e] Negative knowledge cannot be negated
[f] Inactivation of certificates is achieved deductively
[g] Negation of certificates is achieved deductively

Table 5.1. Interactive rules

5.10 Graph Rule Layers

When the graph is in certain states, two or more rules may affect each other's applicability (i.e. they are not independent of each other, see section 3.3). To avoid such situations, we define three graph rule layers. The first layer consists of interactive rules, that can be applied by entities or an administrator. The following layers consist of deductive rules. In the second layer are rules that add information, and in the third layer are rules that remove information. Rule 28 (which neither adds nor removes objects) is in both layer 2 and layer 3, since it may be necessary in both of them to mark the earliest inactivation time. Note that the companion rule 29 only applies after knowledge with an earlier inactivation time has been added, so it is not needed in layer 3.

1. Interactive rules: 1, 2, 3, 4, 6, 7, 13, 16, 19, 24, 26, 39, 41
2. Deductive adding rules: 5, 8, 9, 10, 11, 12, 25, 27, 28, 29, 30, 40, 42, 43
3. Deductive removal rules: 14, 15, 17, 18, 20, 21, 22, 23, 28, 31, 32, 33, 34, 35, 36, 37, 38, 44, 45, 46, 47, 48, 49, 50, 51

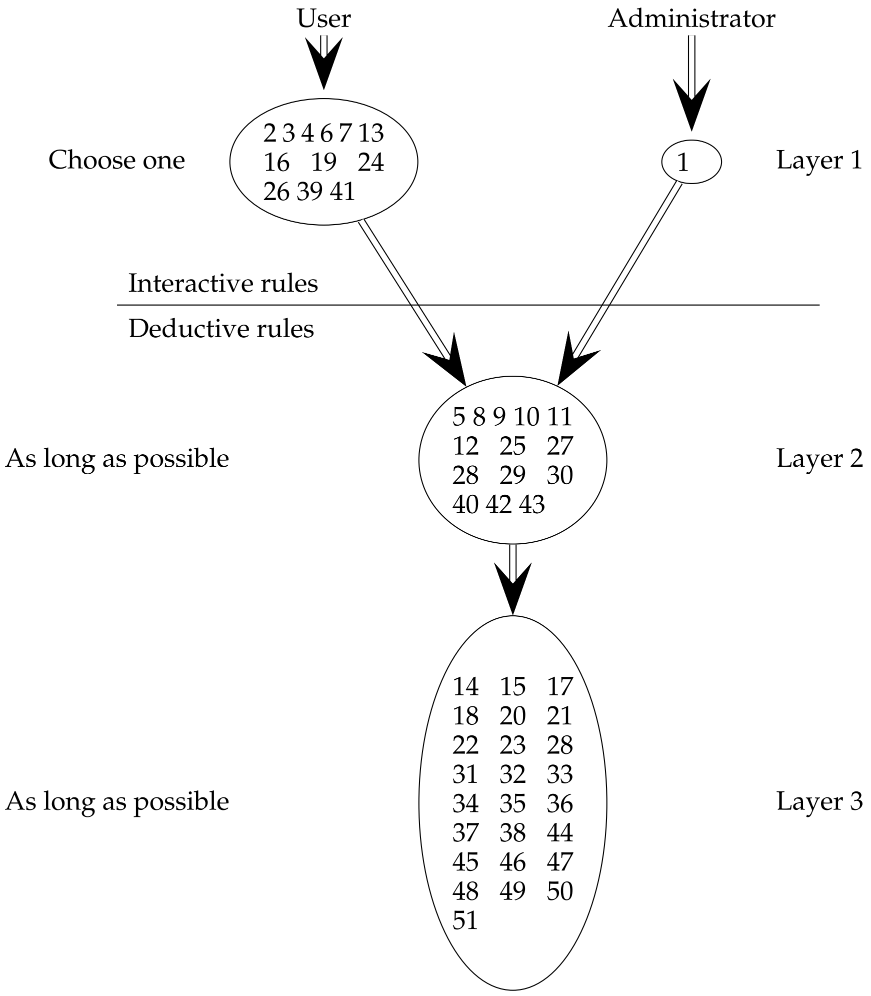


Figure 5.16. The layers of the C-graph rules

Figure 5.16 describes the layers graphically. Users or administrators are allowed to call the rules from layer 1; these are the interactive rules. After the application of one of these rules, the other layers must be passed. The rules of layer 2 will be applied as long as possible, then the rules from layer 3. Within each layer the rules may be applied in any order; they are not in conflict with each other (this is shown in section 6.2.3).

It is important to note that lower layers do not affect the layers above them — i.e., when layer 2 has been passed, no rules in layer 3 make the rules applicable again (this is shown in section 6.2.3). This means that the layering only helps to separate conflicting rules, it does not hinder the flow of revocation in any way.

Chapter 6

Analysis of the C-graph

In this chapter we will analyze the C-graph system: first by studying the flow of actions induced by the different types of revocation, then by proving some formal properties of the C-graph.

6.1 Revocation Algorithms

It may be difficult to get an overview of the rules in the previous chapter. In this section, we illustrate the revocation algorithms in a more comprehensive way, by describing flowcharts of how the different types of revocation propagate in a system.

Figure 6.1 describes the flow of actions and system states that may result from any form of removal of a statement. Ovals describe groups of one or more rule actions. The gray ovals represent actions that may be interactive (and may thus be entry points into the flowchart); white ovals describe changes that are purely deductive. The dashed lines that leave each oval point to boxes decribing states — problematic scenarios that may or may not apply after the action in the oval has been taken — note that several may apply. After each box, a solid line points to the action that will be taken in the C-graph to resolve the problem in the box. The flow of actions may end after any action, when no problem state applies.

Rule numbers have been omitted for legibility, but note that each oval may represent a group of possible rules which all perform the named action.

Note that some problem boxes describe states for other keys than the one that was just removed. When moving from such a box to an oval, the action taken will affect another key than the one that was just removed.

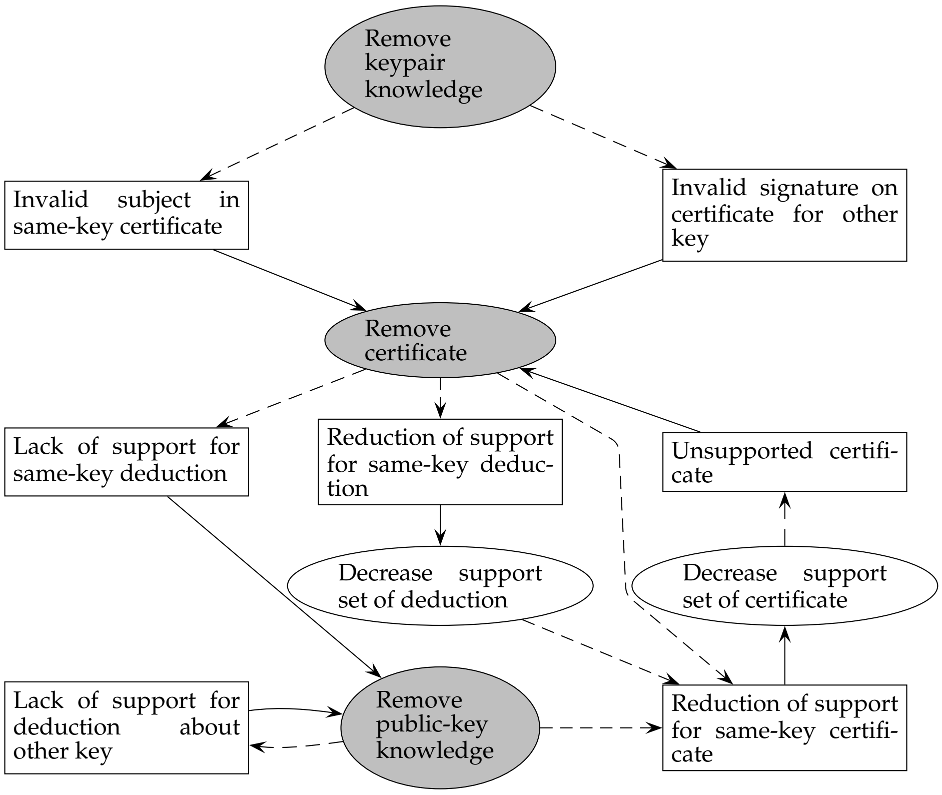

Remove keypair knowledge
Invalid subject in same-key certificate
Invalid signature on certificate for other key
Remove certificate
Lack of support for same-key deduction
Reduction of support for same-key deduction
Unsupported certificate
Decrease support set of deduction
Decrease support set of certificate
Lack of support for deduction about other key
Remove public-key knowledge
Reduction of support for same-key certificate

Figure 6.1. Revocation by removal

As an example, assume that an entity has just applied rule 19 to remove a certificate that they had previously quoted to another entity. This means that the flowchart is entered in the oval marked ”Remove certificate”. In this case, three different problems may arise with the entity that previously received the newly removed certificate. If they have deduced knowledge about the subject key of the certificate, the deduction may have lost its support completely — in which case the leftmost path of the flowchart should be taken — or may have had its support reduced — in which case the middle path applies. If the entity quoted the certificate to somebody else, this quoted certificate may have had its support reduced, in which case the rightmost path in the flowchart applies. Note that several cases may apply, and that in that case, several paths will be taken.

Figure 6.2 describes the flow of actions and system states that may result from an inactivation. In this chart, an added type of ovals is used: the hatched ovals represent actions that are part of the removal flowchart (fig-

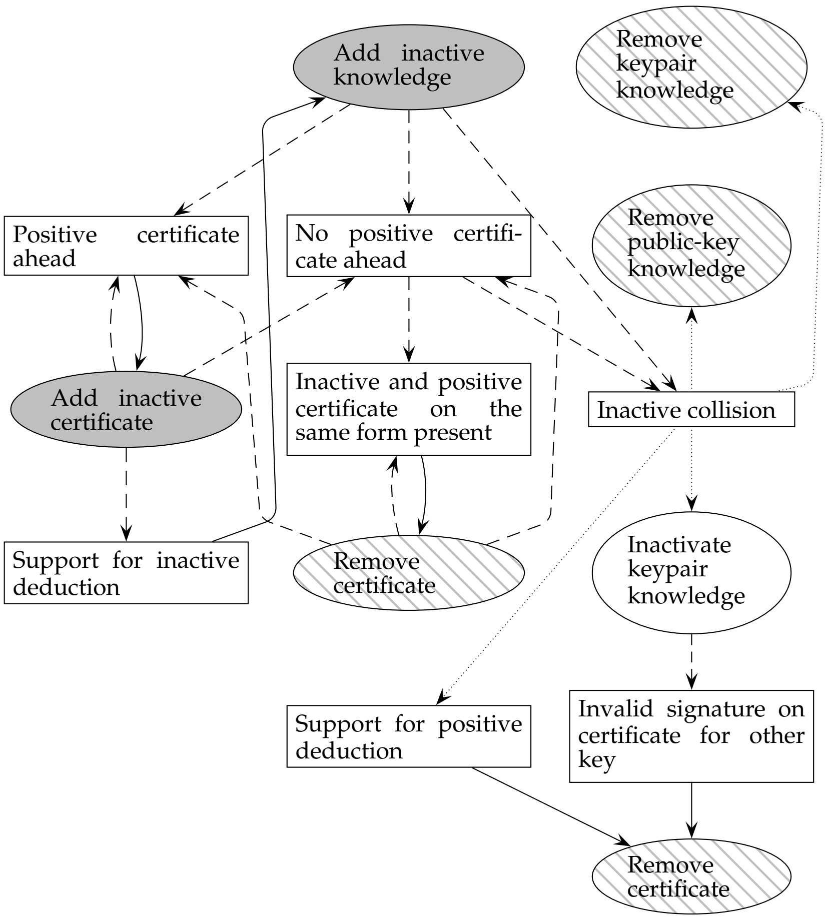

Add inactive knowledge
Remove keypair knowledge
Positive certificate ahead
No positive certificate ahead
Remove public-key knowledge
Add inactive certificate
Inactive and positive certificate on the same form present
Inactive collision
Support for inactive deduction
Remove certificate
Inactivate keypair knowledge
Support for positive deduction
Invalid signature on certificate for other key
Remove certificate

Figure 6.2. Revocation by inactivation

ure 6.1). When one of these ovals are reached, the flow of actions jumps to the corresponding oval in the removal flowchart, and continues from there.

In this chart, the state box marked ”Inactive collision” has several dotted lines leaving it. These describe alternative ways to solve the problem of the collision, depending on the reason for it — note that at least one of the alternatives must apply.

The state boxes marked ”Positive certificate ahead” and ”No positive certificate ahead”, and the actions that follow them, represent the propagation/backtracking functionality.

Figure 6.3, finally, describes the flow of actions and system states that may result from a negation. It is analogous in structure to the inactivation

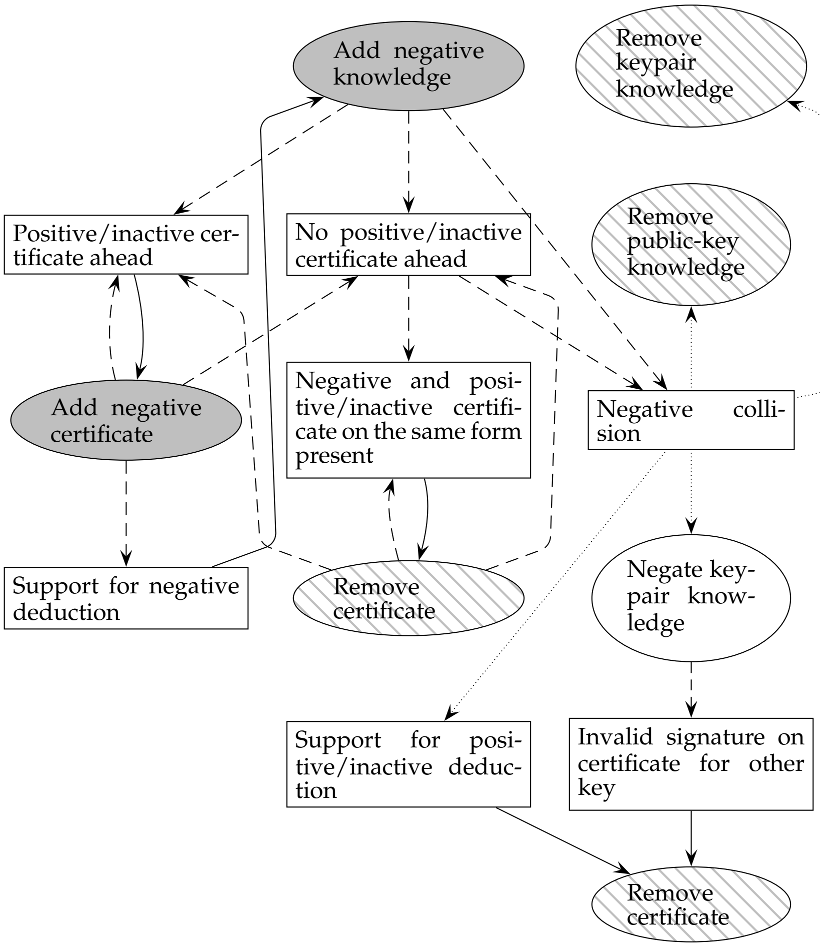

Add negative knowledge
Remove keypair knowledge
Positive/inactive certificate ahead
No positive/inactive certificate ahead
Remove public-key knowledge
Add negative certificate
Negative and positive/inactive certificate on the same form present
Negative collision
Support for negative deduction
Remove certificate
Negate keypair knowledge
Support for positive/inactive deduction
Invalid signature on certificate for other key
Remove certificate

Figure 6.3. Revocation by negation

flowchart.

Studying the flowcharts, it is interesting to note that removal of a statement may lead to the removal of statements about other keys, whereas inactivation and negation of a statement never can lead to inactivation or negation of other keys, respectively. However, inactivation and negation may both induce removal of statements about the inactivated/negated key as well as removal of statements about other keys.

The flowcharts describe how the C-graph rules work in concert, but the rules may also be viewed as an implementation of the flowcharts. In other words, the flowcharts represent what we believe to be appropriate revocation practices for a system with the properties described in chapter 4.

6.2 C-graph Properties

We will now investigate the formal properties of the C-graph model. In particular, we will establish soundness and completeness results for the graph rules from chapter 5 with respect to the notion of a valid state, which we recapitulate:

Definition 13 (Valid state)**.** *A system is in a **valid state** if and only if:*

(1) the support set of each statement is valid, complete and non-empty;

(2) there are no collisions;

(3) where there is support for deduced knowledge, the deduction is made;

(4) when an entity has one or more inactive public-key knowledge statements about a specific binding, the one with the earliest inactivation time is marked.

We define *validity* and *stability* for the C-graph, two concepts that are necessary in the following discussion:

Definition 15 (Validity)**.** *A C-graph is said to be **valid** if it represents a valid state (see definition 13).*

Definition 16 (Stability)**.** *A C-graph is said to be **stable** if no deductive rule (5, 8, 9, 10, 11, 12, 14, 15, 17, 18, 20, 21, 22, 23, 25, 27, 28, 29, 30,31, 32, 33, 34, 35, 36, 37, 38, 40, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51) applies.*

In order to prove soundness and completeness for the C-graph rules, we will need a few preliminary results:

Observation 1 (Existing subjects)**.** *In a C-graph, the subject of every statement is an existing entity.*

Proof. Rules 2 and 3 are the only rules that add new knowledge to the system. In both cases, the subject of the knowledge is an existing entity, and no rules remove entities. □

Lemma 1 (Support sets are valid and complete)**.** *In a stable C-graph generated by the rules in chapter 5, the support set of any statement σ is valid, complete, and non-empty.*

Proof. Assume that a given C-graph, generated by the rules in chapter 5, is stable. Consider the support set Ω of a statement σ.

- Assume that σ is a keypair knowledge. The support set of σ was generated by rule 2 when σ was created — at this time, Ω was valid, complete, and non-empty. No rules change the support or support set of keypair knowledge, so Ω must be valid, complete, and non-empty.

- Assume that σ is a type O public-key knowledge. The support set of σ was generated by rule 3 when σ was created — at this time, Ω was valid, complete, and non-empty. No rules change the support set of type O public-key knowledge, so Ω must be valid, complete, and non-empty.

- Assume that σ is a type I public-key knowledge. Rules 9 and 11 ensured that Ω was valid, complete, and non-empty at the time σ was added, given that its supporting certificate had a valid and complete support set. When any supporting certificate is added, rules 10 and 12 ensure that all (and only) new valid paths are added to Ω, given that the new support has valid and complete support sets. When any supporting certificate is removed, rules 22 and 23 ensure that all (and only) the missing support is removed from Ω, and rule 18 removes σ itself if all support is missing.

- Assume that σ is a certificate. Rules 4, 6 and 7 ensured that Ω was valid, complete, and non-empty at the time σ was added, given that its supporting statement had a valid and complete support set. When any supporting statement is added, rules 5 and 8 ensure that all (and only) new valid paths are added to Ω, given that the new support has valid and complete support sets. When any supporting statement is removed, rules 17 and 20 ensure that all (and only) the missing support is removed from Ω, and rules 14, 15, and 21 remove σ itself if all support is missing.

Since rules 2 and 3 are the only rules that add new information to the graph — with valid and complete support sets — and since that information is in the form of keypair knowledge or type O public-key knowledge, every chain of certificates and deduced knowledge must stem from eitehr of these two types of knowledge. By observation 1, the subject of every statement in the C-graph is an existing entity. Hence, by induction, each support set in the graph must be valid, complete, and non-empty. $\square$

Lemma 2 (Freedom from collisions)**.** *In a stable C-graph, there are no collisions.*

Proof. Assume that a given C-graph is stable, i.e. that no deductive rule applies.

Assume that there is an inactive collision between the two knowledge statements σ (positive) and ρ (inactive). Note that the entity holding the conflicting knowledge either distributes a certificate for the positive knowledge, or not. One of the following cases must apply:

- If σ and ρ are both keypair knowledge statements, either rule 25 or rule 33 applies, and hence the C-graph is not stable.
- If σ is a keypair knowledge, and ρ is a public-key knowledge, either rule 27 or rule 32 applies, and hence the C-graph is not stable.
- If σ is a public-key knowledge, and ρ is a keypair knowledge, either rule 25, rule 37 or rule 38 applies (in this case, ρ must be of type I, so there must be support for it), and hence the C-graph is not stable.
- If σ and ρ are both public-key knowledge statements, either rule 27 or rule 34 applies, and hence the C-graph is not stable.

Alternatively, assume that there is a negative collision between the two knowledge statements σ (positive or inactive) and ρ (negative). One of the following cases must apply:

- If σ and ρ are both keypair knowledge statements, either rule 40 or rule 46 applies, and hence the C-graph is not stable.
- If σ is a keypair knowledge, and ρ is a public-key knowledge, either rule 42 or rule 45 applies, and hence the C-graph is not stable.
- If σ is a public-key knowledge, and ρ is a keypair knowledge, either rule 40, rule 50 or rule 51 applies (in this case, ρ must be of type I, so there must be support for it), and hence the C-graph is not stable.
- If σ and ρ are both public-key knowledge statements, either rule 42 or rule 47 applies, hence the C-graph is not stable.

Hence, a stable C-graph cannot include certificate collisions. □

6.2.1 Soundness

We will now show soundness of the C-graph formalism, i.e., that stable C-graphs generated by the given rules are valid. After applying any of the interactive rules, the system will apply the deductive rules as long as they are applicable. The soundness theorem shows that if the system reaches a stable state, the state is also guaranteed to be valid.

Theorem 1 (Soundness)**.** *Any stable C-graph generated by the rules in chapter 5 is valid.*

Proof. Assume that a given C-graph generated by the rules in chapter 5 is stable, and that it is not valid (i.e., at least one of the properties of definition 13 is not satisfied).

- If property (1) is not satisified, there exists a statement with a support set that is not valid, complete, and non-empty. By lemma 1, the C-graph cannot be stable.
- If property (2) is not satisfied, there exists an inactive or negative collision in the graph. By lemma 2, the C-graph cannot be stable.
- If property (3) is not satisfied, there is support for a deduction which is not made. Hence, rule 9, 10, 11 or 12 applies, and hence the C-graph is not stable.
- If property (4) is not satisfied, there is an entity with inactive knowledge, where the knowledge with the earliest inactivation time is not marked. Hence, either rule 28 or rule 29 applies, and hence the C-graph is not stable.

It follows that an invalid C-graph cannot be stable, and hence a stable C-graph must be valid. □

6.2.2 Completeness

Completeness for the C-graph means that all valid states are stable, and can be generated by the given rules.

Theorem 2 (Completeness)**.** *Any valid C-graph is stable and can be generated by the rules in chapter 5.*

Proof. Assume that a given C-graph is valid but not stable, i.e., one of the deductive rules applies:

- If rule 5 or 8 applies, there exists a certificate with an incomplete support set, i.e., property (1) of the valid state is violated.
- If rule 10 or 12 applies, there exists a knowledge statement with an incomplete support set, i.e., property (1) of the valid state is violated.
- If rule 14, 15, 17, or 20 applies, there exists a certificate with a non-valid support set, i.e., property (1) of the valid state is violated.
- If rule 18, 22, or 23 applies, there exists a knowledge statement with a non-valid support set, i.e., property (1) of the valid state is violated.
- If rule 21 applies, there exists a certificate with an empty support set, i.e., property (1) of the valid state is violated.
- If rule 25 or 27 applies, there is either support for the positive certificate in the left-hand side of the rule or not. If there is support, X has an inactive collision, i.e., property (2) of the valid state is violated. If there is not support, property (1) of the valid state is violated.
- If rule 28 or 29 applies, then the earliest inactivation time is not marked, hence property (4) of the valid state is violated.
- If rule 30 or 31 applies, then either X has an inactive collision and property (2) of the valid state is violated, or one of the certificates on the left-hand side of the rule has a non-valid support set, hence property (1) of the valid state is violated.
- If rule 32, 33, or 34 applies, there exists an entity with an inactive collision, hence property (2) of the valid state is violated.
- If rule 35, 36, 37, or 38 applies, there is either support for a deduction that has not been made and property (3) of the valid state is violated, or there is an inactive collision and property (2) of the valid state is violated.
- If rule 40 or 42 applies, there is either support for the positive/inactive certificate in the left-hand side of the rule or not. If there is support, X has a negative collision, i.e. property (2) of the valid state is violated. If there is not support, property (1) of the valid state is violated.
- If rule 43 or 44 applies, then either X has a negative collision and property (2) of the valid state is violated, or one of the certificates

on the left-hand side of the rule has a non-valid support set, hence property (1) of the valid state is violated.

- If rule 45, 46, or 47 applies, there exists an entity with a negative collision, hence property (2) of the valid state is violated.
- If rule 48, 49, 50, or 51 applies, there is either support for a deduction that has not been made and property (3) of the valid state is violated, or there is a negative collision and property (2) of the valid state is violated.

Hence, a valid C-graph must be stable.

Now consider the rules used to generate the valid C-graph.

First, the interactive rules: Any entities needed in the system can be generated by rule 1. Any keypair knowledge can be generated by rule 2, and any outside knowledge can be added by rule 3. Any supported certificates can be generated by rule 4 and 6, and quoted by rule 7. Any keypair knowledge can be removed by rule 13, any type O public-key knowledge can be removed by rule 16. Any certificate may be removed by rule 19. Any keypair knowledge can be inactivated by rule 24, and any type O public-key knowledge can be inactivated by rule 26. Any keypair knowledge can be negated by rule 39, and any type O public-key knowledge can be negated by rule 41.

Next, the deductive adding rules: If a certificate has an incomplete support set, rule 5 or 8 will add the necessary paths. If a type I public-key knowledge has an incomplete support set, rule 10 or 12 will add the necessary paths. If there is support for a deduction, rule 9 or 11 will make the deduction. If an entity has inactive public-key knowledge for a key with an earlier inactivation time than any other inactive knowledge that the same entity has about the same key, and it is not marked (because it is newly added), then rule 28 will mark that knowledge. If an entity has a marked inactive public-key knowledge for a key, but that knowledge has a later inactivation time than some other inactive knowledge that the same entity has about the same key (because knowledge with an earlier inactivation time is newly added), rule 29 will remove the incorrect mark.

Finally, the deductive removal rules: If a certificate has a non-valid support set, rule 14, 15, 17, or 20 will either remove the certificate if it lacks support completely, or update the support set. If a certificate has an empty support set, rule 21 will remove it. If a type I public-key knowledge has a non-valid support set, rule 18, 22, or 23 will either remove the knowledge if it lacks support completely, or update the support set. If an entity has inactive public-key knowledge for a key with an earlier inactivation time than any other inactive knowledge that the same entity has about the same key, and it is not marked (because the knowledge that was previously marked has been removed), then rule 28 will mark that knowledge.

Hence, the interactive rules allow any supported statements to be issued and distributed in the C-graph. The deductive rules ensure that the properties of the valid state are all observed. All statements have valid, complete, and non-empty support sets, as required by property (1). The C-graph is stable, so by lemma 2, it is free of certificate collisions, as required by property (2). All supported deductions are made, as required by property (3), and all inactive knowledge with the earliest inactivation time for a specific key held by an entity is marked, as required by property (4). Hence, any valid C-graph can be generated by the rules in chapter 5. □

6.2.3 Rule Independence

As specified in section 5.10, the C-graph rules are separated into three layers: interactive rules, deductive adding rules, and deductive removal rules. In this section we will examine the layers of deductive rules and show parallel and sequential independence within and between them, respectively. It is important to remember that each pass of the rules begins with the application of exactly one of the interactive rules; hence, it is for example impossible for two different keys to be inactivated in the same round.

Parallel Independence

Parallel independence between two derivations ensures that they may occur in any order with the same result. We recall the requirements for parallel independence between two derivations [EHK+97]:

(1) neither derivation deletes an object that is necessary for matching the other, and

(2) neither derivation establishes a context that is forbidden by a NAC of the other.

For parallel independence between two rules, all possible matchings of the rules must be parallel independent. To convince ourselves of the parallel independence between the rules in each layer, we have compared each pair of rules with respect to the requirements above.

Table 6.1 shows the analysis of layer 2. When it comes to adding rules, the first requirement is never a problem, since no rules delete any objects. The second requirement, however, must be considered. There are some cases when the requirement is not met, but closer examination shows that these cases cause no problems. In particular, two rules that add the same object obtain identical results; it does not matter which rule is the one to perform the operation.

Tables 6.2 and 6.3 show the analysis of layer 3. For deductive removal rules, the second requirement of parallel independence is never a problem — since no rules add new objects — but the first requirement must be considered. These rules have more conflicts, but when examined they can be shown not to cause problems (see the footnotes for each table). Some rules may remove the same object, but it does not matter which rule performs the operation.

Rules	5	8	9	10	11	12	25	27	28	29	30	40	42	43
43	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
42	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	
40	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2		
30	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2			
29	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2				
28	1,2	1,2	1,2[a]	1,2	1,2[a]	1,2	1,2	1,2	1,2					
27	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2						
25	1,2	1,2	1,2	1,2	1,2	1,2	1,2							
12	1,2	1,2	1,2	1,2[b]	1,2	1,2								
11	1,2	1,2	1,2[b]	1,2	1,2									
10	1,2	1,2	1,2	1,2										
9	1,2	1,2	1,2											
8	1,2	1,2												
5	1,2													

[a] Rule 29 solves the conflict that may arise in rule 28 if new knowledge is deduced
[b] These rules may add the same object

Table 6.1. Parallel independence of deductive adding rules

Rules	14	15	17	18	20	21	22	23	28	31	32	33
51	1[i],2	1,2	1[b],2	1[i],2	1[b],2	1,2	1[f],2	1[f],2	1,2	1[j],2	1,2	1,2
50	1[i],2	1,2	1[b],2	1[i],2	1[b],2	1,2	1[f],2	1[f],2	1,2	1[j],2	1,2	1,2
49	1[i],2	1,2	1[b],2	1[i],2	1[b],2	1,2	1[f],2	1[f],2	1,2	1[j],2	1,2	1,2
48	1[i],2	1[i],2	1[b],2	1[i],2	1[b],2	1[c],2	1,2	1,2	1,2	1[j],2	1,2	1,2
47	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1[h],2	1,2	1,2	1,2
46	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1[j],2
45	1,2	1,2	1,2	1[d],2	1,2	1,2	1,2	1,2	1,2	1,2	1[j],2	1,2
44	1[i],2	1[a],2	1[b],2	1,2	1[b],2	1[c],2	1,2	1,2	1,2	1[j],2	1,2	1,2
38	1[g],2	1,2	1[b],2	1[g],2	1[b],2	1,2	1[f],2	1[f],2	1,2	1[c],2	1,2	1,2
37	1[g],2	1,2	1[b],2	1[g],2	1[b],2	1,2	1[f],2	1[f],2	1,2	1[c],2	1,2	1,2
36	1[g],2	1,2	1[b],2	1[g],2	1[b],2	1,2	1[f],2	1[f],2	1,2	1[c],2	1,2	1,2
35	1[g],2	1[g],2	1[b],2	1[g],2	1[b],2	1[c],2	1,2	1,2	1,2	1[c],2	1,2	1,2
34	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
33	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
32	1,2	1,2	1,2	1[e], 2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	
31	1[g],2	1[a],2	1[b],2	1,2	1[b],2	1[c],2	1,2	1,2	1,2	1,2		
28	1,2	1,2	1,2	1[h],2	1,2	1,2	1,2	1,2	1,2			
23	1,2	1,2	1,2	1[f],2	1,2	1,2	1,2	1,2				
22	1,2	1,2	1,2	1[f],2	1,2	1,2	1,2					
21	1,2	1,2	1,2	1,2	1,2	1,2						
20	1,2	1,2	1,2	1,2	1,2							
18	1,2	1,2	1,2	1,2								
17	1[b],2	1,2	1,2									
15	1[c],2	1,2										
14	1,2											

[a] The propagation/backtracking rules ensure that these rules do not conflict
[b] If the certificate is removed, it is unnecessary to update the support set
[c] These rules may remove the same object
[d] Keys cannot be both negated and removed during the same round
[e] Keys cannot be both inactivated and removed during the same round
[f] If the knowledge is removed, it is unnecessary to update the support set
[g] Different keys cannot be inactivated during the same round
[h] If the knowledge is removed, it is unnecessary to mark it
[i] Different keys cannot be negated during the same round
[j] Keys cannot be both inactivated and negated during the same round

Table 6.2. Parallel independence of deductive removal rules, part 1

Rules	34	35	36	37	38	44	45	46	47	48	49	50	51
51	1,2	1,2	1,2	1,2	1,2	1[c],2	1,2	1,2	1,2	1[c],2	1[c],2	1[c],2	1,2
50	1,2	1,2	1,2	1,2	1,2	1[c],2	1,2	1,2	1,2	1[c],2	1[c],2	1,2	
49	1,2	1,2	1,2	1,2	1,2	1[c],2	1,2	1,2	1,2	1,2	1[c],2		
48	1,2	1[b],2	1,2	1,2	1,2	1[c],2	1,2	1,2	1,2	1,2			
47	1[b],2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2				
46	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2					
45	1,2	1,2	1,2	1,2	1,2	1,2	1,2						
44	1,2	1[b],2	1[j],2	1[j],2	1[j],2	1,2							
38	1,2	1[d],2	1[c],2	1,2	1,2								
37	1,2	1[d],2	1[c],2	1,2									
36	1,2	1[d],2	1,2										
35	1,2	1,2											
34	1,2												

[a] If the knowledge is removed, it is unnecessary to mark it with an asterisk
[b] Keys cannot be both inactivated and negated in the same round
[c] These rules may remove the same object

Table 6.3. Parallel independence of deductive removal rules, part 2

Sequential Independence

A derivation d_2' is weakly sequentially independent of d_1 if it does not rely on d_1 to be applied before it. As stated in chapter 3, the following requirements ensure that the derivation d_2' is weakly sequentially independent of d_1 [EHK$^+$97]:

(1) the overlapping of the right-hand side of d_1 and the left-hand side of d_2' does not contain elements which are generated by d_1, and

(2) no context that is forbidden by a NAC of d_2' is destroyed by d_1.

Rules	5	8	9	10	11	12	25	27	29	30	40	42	43
14	1,2	1,2	1,2	1,2	1,2	1,2	1,2[a]	1,2[b]	1,2	1,2[b]	1,2[a]	1,2[b]	1,2[b]
15	1,2	1,2	1,2	1,2	1,2	1,2	1,2[a]	1,2[b]	1,2	1,2[b]	1,2[a]	1,2[b]	1,2[b]
17	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
18	1,2	1,2	1,2[a]	1,2	1,2[a]	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
20	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
21	1,2	1,2	1,2	1,2	1,2	1,2	1,2[a]	1,2[a]	1,2	1,2[a]	1,2[a]	1,2[a]	1,2[a]
22	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
23	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
31	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
32	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
33	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
34	1,2	1,2	1,2[d]	1,2	1,2[d]	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
35	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
36	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
37	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
38	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2[c]
44	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2	1,2[c]	1,2	1,2	1,2
45	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
46	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
47	1,2	1,2	1,2[f]	1,2	1,2[f]	1,2	1,2	1,2	1,2	1,2	1,2	1,2	1,2
48	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2	1,2[c]	1,2	1,2	1,2
49	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2	1,2[c]	1,2	1,2	1,2
50	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2	1,2[c]	1,2	1,2	1,2
51	1,2	1,2	1,2	1,2	1,2	1,2	1,2[c]	1,2[c]	1,2	1,2[c]	1,2	1,2	1,2

[a] These rules are mutually exclusive
[b] Different keys cannot be inactivated during the same round
[c] Keys cannot be both inactivated and negated during the same round
[d] Rule 35 ensures that these two rules are mutually exclusive
[e] Different keys cannot be negated during the same round
[f] Rule 48 ensures that these two rules are mutually exclusive

Table 6.4. Weak sequential independence of layer 2 with respect to layer 3

For the C-graph, we must show that the rules (i.e. all possible matchings) in layer 2 are weakly sequentially independent of the rules in layer 3. The analysis is shown in table 6.4. Since rule 28 is in both layers, it is not necessary to show sequential independence for that rule.

Since rules in layer 3 do not generate any elements, the first requirement is trivially met for all pairs of rules in the table. The second requirement needs more examination, but each case of potential conflict can be shown to be nonproblematic.

6.2.4 Summary

We have showed that the rules in chapter 5 can generate any valid C-graphs, and that validity and stability for the C-graph occur simultaneously. This means that when the rules have been applied as long as possible the result will be a valid system, and that when a valid state is reached, no rules will apply.

The parallel independence of the rules in their respective layers guarantees that the order in which the rules are applied has no effect on the result of their application, i.e., the state of the C-graph after a layer has reached stability will be deterministic. The sequential independence between the layers guarantees that no removal rule will make an adding rule applicable, i.e., that one pass through the rule layers is sufficient for the C-graph to reach a valid and stable state.

Chapter 7

Model Evolution

In this chapter we will give an overview of how the C-graph model has developed along with motivations for some of the choices that were made along the way, as well as suggestions for future extensions to the model.

7.1 Development of the C-graph Model

The C-graph model has developed over time. In this section we will give some examples of how it has changed and why, regarding time stamps and how statements are represented.

Originally, all edges were on the form of certificates with only two fields: one for a signing key identifier, one for a public key (being certified). This abstraction was used both for entities' internal knowledge of keys, and for quotations passed between users. An entity A's knowledge about a user B's key was modeled as a certificate held by A, signed with one of A's own keys. The structure for entities' knowledge about their own keys was unclear. The rule for adding new knowledge looked like in figure 7.1, with just a NAC for avoiding duplicates.

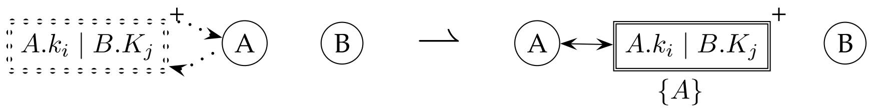


Figure 7.1. Adding knowledge

In this early version, inactivation was solved by a rule expression which inactivated all certificates for a key in the entire system at once. When the localness assumptions in chapter 4 were formulated, it became clear that simultaneously inactivating all certificates for a key was in conflict with these basic assumptions of how the system should work. We decided on local inactivations which were spread in the system in the same way as negative certificates. This meant that users would not necessarily know about the inactivation of one of their keys instantly, but may continue to issue certificates due to ignorance until information about the inactivation finally reached them. Time stamps would have to be introduced, to keep track of when a certificate was issued, and when a key was inactivated.

Certificates were still used to model all kinds of statements, but we started to think about what would happen if the key A used to sign a certificate with was inactivated by some other entity. Since A only has local knowledge, the only case where the inactivation mattered was if A knew about it. This gave us a new variant of the rule for adding knowledge, shown in figure 7.2.

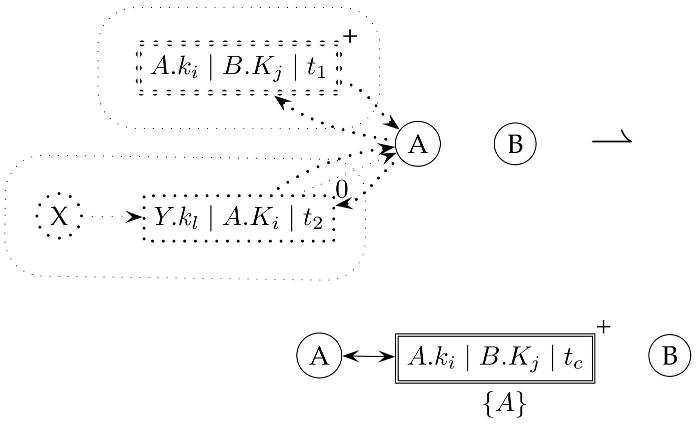

$A.k_i \mid B.K_j \mid t_1$
+
A
B
X
$Y.k_l \mid A.K_i \mid t_2$
0
A
$A.k_i \mid B.K_j \mid t_c$
+
$\{A\}$
B

Figure 7.2. Adding knowledge

However, A should have knowledge about Y's public key $Y.K_l$ to be able to check the signature on the incoming certificate, and this is where we ran into trouble. What if A's knowledge of $Y.K_l$ had been signed with $A.k_i$, the very key that was being inactivated? Would we accept that knowledge now that $A.k_i$ was inactivated? Furthermore, why would A have to choose only one of their keys to sign that knowledge? If it happened to have been signed with $A.k_i$, it would be problematic to accept Y's certificate, a problem that would not have arisen, had the knowledge been signed with another of A's keys. This indicates a form of non-determinism and haphazardness in the system.

Additionally, even if the knowledge of $Y.K_l$ had been signed with some other key $A.k_x$, how would we be sure that $A.k_x$ was valid? By checking if that key in turn had been signed by another key that was valid? This created a chicken-and-egg problem, which ultimately was due to the fact that we used one and the same data structure for all information in the system.

To avoid these problems, we first thought about assigning to each entity an internal key k_0 that they would use to sign their own internal knowledge. In this scenario, a rule for adding knowledge might look like in figure 7.3.

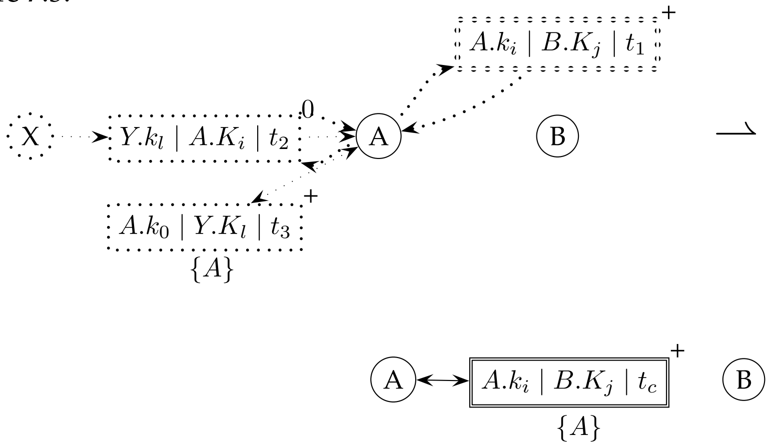

$A.k_i \mid B.K_j \mid t_1$
+
X
$Y.k_l \mid A.K_i \mid t_2$
0
A
B
$A.k_0 \mid Y.K_l \mid t_3$
+
$\{A\}$
A
$A.k_i \mid B.K_j \mid t_c$
+
B
$\{A\}$

Figure 7.3. Adding knowledge

This approach solved the chicken-and-egg problem as well as the randomness that choosing any key to sign knowledge attributed, but unfortunately it was not an ideal solution. If the internal key k_0 of each entity was really to be an internal key, it should not be used when quoting certificates to other entities. But if we were to keep the certificate structure for internal knowledge, the rules of the system would apply to the knowledge certificates as well — so, for instance, they would be quotable. This meant that every rule would need a special condition attached, specifying that signing keys should not be internal keys.

Another approach was to accept that it is not necessary to represent knowledge by certificates, using instead another data structure that does not include a signature. Entities can have knowledge about their own public-key pairs (which they are assumed to generate themselves), and about other entities' public keys. Knowledge about other entities' keys can be deduced from other information, or formed outside the system. Therefore, we need two other kinds of boxes beside the certificates: key-pair knowledge and public-key knowledge, as described in chapter 5.

A final adjustment was made to the time stamps when we realized that the inactivation time must follow all statements about the inactivated key, but the signing time of a certificate must also be included. A certificate for the inactivated key $B.K_j$ signed with $A.k_i$ must keep track of $B.K_j$'s inactivation time, as well as the signing time, if $A.k_i$ should ever be inactivated. Therefore, an additional time stamp was appended to the certificate structure.

7.2 Model Extensions

We will now consider possible extension to the rules of chapter 5, by outlining three ways to revoke a user, as well as the addition of trust statements.

7.2.1 User Revocation

In chapter 5, keys were revoked one at a time. A more efficient way of revoking all the keys of a user would be to revoke the user itself. To begin, we would need to expand the vertice attribution function v-att to be able to express that a user is inactivated or negated:

$$\text{v-att} = \begin{cases} \text{name}: & V \to \text{V-ATT} \\ \text{v-sign}: & V \to \{+, 0, -\} \end{cases}$$

In a non-static environment, users may need to be permanently removed from the system, e.g. if they leave their jobs and any information produced by them is no longer needed. Removal of a user node would be an administrative, active rule, shown in figure 7.4.

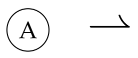


Figure 7.4. Removing an entity

When a user is removed from the graph, it is necessary to clean up all information that either stems from or relates to that user. An important reason for this is to ensure that each subject or signer of a certificate corresponds to an existing user. First consider certificates signed by the user: since graph transformation rules automatically remove dangling edges when a node is removed, all original knowledge that had been signed by the removed user will be removed. Following this, the revocation rules of section 5.6 will revoke all certificates that are no longer supported — i.e. all quoted certificates with a signature by the removed user, and deductions based on such certificates. Next, consider certificates signed by others, but quoted by the user: these will also be removed by the dangling-edge clean-up, and again the rules from section 5.6 will complete the revocation.

Finally consider certificates for the removed user, i.e. where the user is the subject. Here we run into a situation that is unclear in our current model: do users know the identity and status of other users? If they do, it would be easy to construct a rule which removes statements about removed users — see figure 7.5 — but if they do not, it seems necessary to include knowledge about other users' identities, possibly as a third variety of knowledge box. This would allow users to hold local knowledge about each other's identities and status. The addition of such knowledge would allow entities, not only administrators, to perform user revocations.

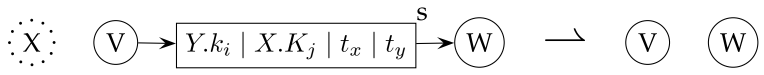


Figure 7.5. Removal of certificates for removed users

The result of the outlined process — which is clearly a form of revocation — is that no information relating to the removed user remains in the C-graph.

Another type of user revocation to be considered is inactivation of a user, where all of a user's keys should be inactivated. This could be used when a user is promoted and receives a new set of keys, but the previous signatures should still be valid. Negation of a user would entail the negation of all the user's keys. This could be useful when a user turns out to have malicious intentions, or even to be a criminal.

7.2.2 Modeling Keys and Trust

In this work we have assumed that all entities are trustworthy, and the rules reflect this assumption. One obvious way to extend the C-graph model would be to depart from the assumption that all users are trustworthy, and to introduce an explicit way for users to talk about trust. Previous research in this field and proposed models have been presented by Blaze et al. [BFL96], Abdul-Rahman and Hailes [ARH97], Maurer et al. [UM96, KM00], Caronni [GC00], Liu et al. [LOC01], Carbone et al. [CNS03], Dragovic et al. [DHH$^+$03] and in OpenPGP [OpenPGP05].

By introducing trust statements, users can specify which users they trust, and to what extent. Knowledge would only be deduced based on a received certificate if the recipient holds the key to verify the signature, and trusts the signer of the certificate. Trust statements could also be removed, inactivated, or negated, as necessary.

7.2.3 Additional Statements

An extension that seems very natural to introduce would be to let entities use their keys for encryption, as well as for signatures. Since the C-graph already includes all the rules necessary for distributing and revoking the keys themselves, all that is needed are rules for encryption and decryption, as well as policies to deal with encrypted messages when a key is revoked in any way, that would work in concert with the existing system.

An interesting and easy-to-implement extension to the model would be allowing any form of signed statements, e.g. tickets, capabilities and delegation of rights.

Chapter 8

Discussion

In this chapter we will discuss some aspects of the C-graph model: using the model, classification of the revocation types, and how paths in our model compare to some related work.

8.1 Using the Model

The C-graph model should be viewed as an attempt to describe how revocation by removal, inactivation, and negation should affect the information that each entity in a distributed system holds. The choices we have made have been stated in chapters 4 and 5. With other choices, e.g. on what constitutes a valid system, what information the entities have access to, and how inactivation and negation shold be spread in the system, other results may have been reached.

The model can be used in two ways: either as a comparison to existing PKI systems when analyzing their methods of revocation, or as a starting point for designing a new PKI model. When doing so, some of the choices of our model may prove to be impractical or even impossible to achieve in reality. For example, we assume that the system only accepts the application of one interactive rule at a time and then takes the time to stabilize before allowing another interactive rule. It would be very difficult to obtain this functionality in real life. Furthermore, in a real-life distributed system nodes may be disconnected from the rest of the system, and thereby miss information that should be passed to them. This means that provable stability and validity may both be out of reach in a real system.

Although the C-graph model is distributed, it can also be used to model a hierarchical system by imposing meta-conditions on some rules. If a central CA is desired, it would be easy to assign one of the entities as a CA, and to let only that entity hold outside knowledge and issue certificates.

It would also be possible to choose certain parts of the C-graph model and to abstain from others, if needed. For example, the rules for negating statements could be left out completely. If a system where removal of statements is undesirable, leaving out rules 13, 16, and 19 would remove the possibility for interactive removal, while keeping the deductive rules that work in concert with the inactivation and negation rules.

8.2 Revocation Classifications

There are many different reasons why a certificate could be revoked [FL98, DC98, HFPS02, GJ00]; this is why we need different revocation schemes. In chapter 5 we described the revocation mechanisms by local operations, but it is more interesting to consider the entire chain of a revocation. The flowcharts in chapter 6 are an example of how the chains can be viewed; here we will try to classify the mechanisms with the dimensions resilience, propagation, and dominance.

Hagström et al. [HJPPW01] describe and classify revocation mechanisms in an access control system, investigating the consequences of a revocation for a graph as a whole. This approach can be used in our context after translating the C-graph into permissions and grants.

Think of the information that is passed from one user A to another B as permissions, and the information that A has (either in the form of knowledge, or as a received quotation) as a positive grant option (meaning that A has the right to grant permissions to others). This makes sense, because A cannot spread information they do not know about, in the same way that they could not give a permission unless they themselves were granted the right to do so. We use this interpretation to analyze the revocation schemes.

8.2.1 Revocation by Removal

Revocation by removal describes revocation of the signature on a certificate, and is used e.g. when an entity changes affiliation. We find that removal of A's knowledge/received quotation results in a weak global delete operation. The revocation propagates in the system, removing statements that are no longer supported, but only if no other statements remain that do support them — this is why the revocation is of the weak type. It is global because it affects not only the recipient of the first removed statement, and it is a delete operation because it removes statements.

For a PKI with a central CA, there would be no other supporting statements, so when the CA removed its own knowledge of a key, the revocation would propagate in the strong sense — no statements about the revoked key would remain in the system.

8.2.2 Revocation by Inactivation

Inactivation models revocation of the binding between the subject and the public key, and is used for reasons such as supersession or cessation of operation of a key. Inactivation propagates in the system and is time-persistent, so it is a global negative operation. When it comes to dominance, inactive statements dominate positive ones, but not negatives; hence it takes the middle road.

8.2.3 Revocation by Negation

Finally, we consider revocation by negative assertions. This is a revocation of the key itself and is used in cases of key compromise. The operation dominates positive and inactive certificates, propagates in the system, and is time-persistent. Hence, it can be described as a strong global negative operation, the most restrictive of all revocation schemes.

If several users certify a key and one of them revokes his certificate by negating it, it will not affect certificates signed by others. This could be a problem in the case of key compromise, since it would be desirable to block every user's use of the revoked certificate. However, in a decentralized system users have to make decisions based on the information at hand. If there is only one CA then the revocation would indeed be strong, and remove every instance of the certificate.

8.3 Paths in Other Models

The paths that are considered in the path validation process of X.509 ([HFPS02], as described in section 2.2) are not what we call paths in our model. X.509 paths consist of a set of certificates, held by the *same* entity, where the entity has knowledge of the trust anchor's key, and the trust anchor certifies the key of a second entity, who certifies the key of a third entity etc, until the wanted certificate is reached and verified. Paths in our model are certificates passed *between* entities, either through quotations or deductions. Cycle detection becomes an issue in our setting since each entity only has local knowledge. Note that the functionality of X.509 path validation is performed in our system as well, although the existence of support is determined continuously as changes are made, not only at verification time.

Paths in the graph model of Capkun et al. [CBH03] are similar to those of X.509.

Chapter 9

Conclusions

We have presented a graphical framework for reasoning about certificates, called the C-graph model. In the C-graph, entities can explicitly state and revoke knowledge they have about keys of their own and of others, and they can sign this knowledge to create certificates which can be distributed and revoked to and from other entities. Revocation can be performed in three ways: by removal, by inactivation, and by negation.

Statements are of three types: keypair knowledge which entities hold about their own keypairs, public-key knowledge which entities hold about other user's public keys, and certificates which are signed statements about keys.

Public-key knowledge can be formed outside the system through some secure out-of-band procedure, or be deduced from statements received by the entities. When an entity holds a certificate for a binding, and has knowledge of the certificate's signing key, they will deduce knowledge of the binding in the certificate. Each statement in the system that is deduced or quoted from other statements has support in knowledge established outside the system.

Cycles are prevented from being considered as support through the use of support sets, which solve the problem of instant cycle detection. By comparing the values of the paths in the support set of an incoming supporting statement σ to those of a depending statement ρ, it can be seen if σ depends cyclically on ρ. If the support set of σ contains paths with a prefix that is a path from ρ's support set, then there is a cycle.

When positive, inactive, and negative knowledge is allowed in a system, collisions may occur when entities hold conflicting knowledge. In the C-graph, conflicts are resolved by giving top priority to negative knowledge, and priority to inactive knowledge if there is no negative knowledge. The overridden positive or inactive knowledge is removed, and, if necessary, the support for that knowledge is removed.

The functionality of the C-graph is modelled by graph transformation rules — partial graph morphisms that describe how a part of the graph should change when certain conditions are met. Some rules are interactive, and can be applied to the C-graph by entities or administrators, but most of the rules are deductive. These bring the C-graph back to a valid state after the application of one of the interactive rules. The set of rules has been divided into layers in order to prevent conflicts between rules and make sure that the outcome of their application is deterministic. In the top layer are the interactive rules, then come those deductive rules which add elements, and finally those deductive rules which remove elements.

We have proved soundness and completeness of the model with respect to the definition of a valid state. Additionally, we have proved that the rules within each layer are parallel independent of each other, and that the layer with deductive adding rules is weakly sequentially independent of the layer with deductive removal rules. This means that the derivations within a layer may occur in any order, and that the adding rules do not depend on the removal rules to be applied before them.

Our model is global in the sense that it includes the knowledge of all entities in the system, and therefore allows the deduction of new knowledge based on other entities' statements. Nevertheless, it is possible that two users have conflicting knowledge. From this perspective, the model is local.

The model is decentralized, because this is the most general way to capture a PKI. Adapting the model to be hierarchical or centralized would be easy, e.g. by separating the entities into CAs and end users, and allowing only CAs to form outside knowledge.

Interesting topics to study in the future include the addition of trust statements and other types of signed or encrypted statements to the model. It would also be interesting to compare the model to existing PKI systems in order to analyze their revocation procedures, or even to design a new PKI system and try to implement sound revocation practices.

Bibliography

[AGG05] "The AGG Environment: The User Manual". `http://tfs.cs.tu-berlin.de/agg/AGG-ShortManual.ps.zip`, 2005.

[AL02] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison Wesley, November 2002. Second Edition.

[ARH97] A. Abdul-Rahman and S. Hailes. "Using Recommendations for Managing Trust in Distributed Systems". In: *Proceedings of the IEEE Malaysia International Conference on Communication (MICC'97)*, Kuala Lumpur, Malaysia, November 1997.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. "Decentralized Trust Management". In: *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 164–173, May 1996.

[BKPPT05] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. "Termination of High-Level Replacement Units with Application to Model Transformation". *Electronic Notes in Theoretical Computer Science*, Vol. 127, No. 4, April 2005. Proceedings of the Workshop on Visual Languages and Formal Methods.

[BLL00] A. Buldas, P. Laud, and H. Lipmaa. "Acountable Certificate Management using Undeniable Attestations". In: P. Samarati, Ed., *Proceedings of the 7th ACM conference on Computer and Communications Security*, pp. 19–24, Athens, Greece, November 2000.

[BLL02] A. Buldas, P. Laud, and H. Lipmaa. "Eliminating counterevidence with applications to accountable certificate management". *Journal of Computer Security*, Vol. 10, No. 3, pp. 273–296, 2002.

[CBH03] S. Capkun, L. Buttyán, and J.-P. Hubaux. "Self-Organized Public-Key Management for Mobile Ad Hoc Networks". *IEEE Transactions on Mobile Computing*, Vol. 2, No. 1, pp. 52–64, January 2003.

[CDFT05] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. "OpenPGP Message Format". Tech. Rep. RFC 2440bis, IETF OpenPGP Working Group, July 2005.

[CDH+05] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas. "Internet X.509 Public Key Infrastructure: Certification Path Building". Tech. Rep. RFC 4158, IETF X.509 Public Key Infrastructure Working Group (PKIX), September 2005.

[CE04] C. Ellison. "SPKI/SDSI Certificate Documentation". `http://world.std.com/~cme/html/spki.html`, 2004.

[CE96] C. Ellison. "Establishing Identity Without Certification Authorities". In: *Proceedings of the Sixth Annual USENIX Security Symposium*, pp. 67–76, San Jose, July 1996.

[CEE+01] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. "Certificate Chain Discovery in SPKI/SDSI". *Journal of Computer Security*, Vol. 9, No. 4, pp. 285–322, 2001.

[CMR+96] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. "Algebraic Approaches to Graph Transformation, Part I: Double Pushout Approach". Tech. Rep. TR-96-17, Università di Pisa, Dipartimento di Informatica, March 1996.

[CNS03] M. Carbone, M. Nielsen, and V. Sassone. "A Formal Model for Trust in Dynamic Networks". Tech. Rep. BRICS RS-03-4, BRICS, Department of Computer Science, University of Aarhus, January 2003.

[DC98] D. Cooper. "A Closer Look at Revocation and Key Compromise in Public Key Infrastructures". In: *Proceedings of the 21st National Information Systems Security Conference*, pp. 555–565, October 1998.

[DHH$^+$03] B. Dragovic, S. Hand, T. Harris, E. Kotsovinos, and A. Twigg. "Managing Trust and Reputation in the XenoServer Open Platform". In: *Proceedings of the 1st International Conference on Trust Management*, May 2003.

[EFL$^+$99] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. "SPKI Certificate Theory". Tech. Rep. RFC 2693, IETF SPKI Working Group, September 1999.

[EHK$^+$97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. "Chapter 4. Algebraic Approaches to Graph Transformation – Part II: Single Pushout Approach and Comparison with Double Pushout Approach". In: G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*, pp. 247–312, World Scientific, 1997.

[EPT04] H. Ehrig, U. Prange, and G. Taentzer. "Fundamental Theory for Typed Attributed Graph Transformation". In: H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, Eds., *Proceedings of the 2nd International Conference on Graph Transformations*, pp. 161–177, Springer-Verlag, Rome, Italy, oct 2004. volume 3256, Lecture Notes in Computer Science.

[FL98] B. Fox and B. LaMacchia. "Certificate Revocation: Mechanics and Meaning". In: R. Hirschfeld, Ed., *Financial Cryptography: Second International Conference, FC'98*, pp. 158–164, Springer-Verlag, February 1998. volume 1465, Lecture Notes in Computer Science.

[GC00] G. Caronni. "Walking the Web of Trust". In: *Proceedings of the 9th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 153–158, June 2000.

[GJ00] C. A. Gunter and T. Jim. "Generalized Certificate Revocation". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 316–329, 2000.

[HFPS02] R. Housley, W. Ford, T. Polk, and D. Solo. "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile". Tech. Rep. RFC 3280, IETF X.509

Public Key Infrastructure Working Group (PKIX), April 2002.

[HJPPW01] Å. Hagström, S. Jajodia, F. Parisi-Presicce, and D. Wijesekera. "Revocations — a Classification". In: *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 2001.

[HvdM03] J. Y. Halpern and R. van der Meyden. "A logical reconstruction of SPKI". *Journal of Computer Security*, Vol. 11, No. 4, pp. 581–614, 2003.

[JRH00] J. R. Howell. *Naming and sharing resources across administrative boundaries*. PhD thesis, Dartmouth College, Hanover, New Hampshire, May 2000.

[KG00] H. Khurana and V. D. Gligor. "Review and Revocation of Access Privileges Distributed with PKI Certificates". In: *Proceedings of the Security Protocols Workshop*, Cambridge, UK, April 2000.

[KM00] R. Kohlas and U. Maurer. "Confidence Valuation in a Public-Key Infrastructure Based on Uncertain Evidence". In: H. Imai and Y. Zheng, Eds., *Public Key Cryptography 2000*, pp. 93–112, Springer-Verlag, January 2000. volume 1751, Lecture Notes in Computer Science.

[KM99] R. Kohlas and U. Maurer. "Reasoning About Public-Key Certification: On Bindings Between Entities and Public Keys". In: M. Franklin, Ed., *Financial Cryptography: Third International Conference, FC'99*, Springer-Verlag, February 1999. volume 1648, Lecture Notes in Computer Science.

[LF01] N. Li and J. Feigenbaum. "Nonmonotonicity, User Interfaces, and Risk Assessment in Certificate Revocation (Position Paper)". In: *Proceedings of the Fifth International Conference on Financial Cryptography*, pp. 166–177, Springer-Verlag, Grand Cayman, February 2001. volume 2339, Lecture Notes in Computer Science.

[LFG99] N. Li, J. Feigenbaum, and B. N. Grosof. "A Logic-based Knowledge Representation for Authorization with Delegation (Extended Abstract)". In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, July 1999.

[LOC01] C. Liu, M. Ozols, and T. Cant. "An Axiomatic Basis for Reasoning about Trust in PKIs". In: V. Varadharajan and Y. Mu, Eds., *Proceedings of the 6th Australasian Conference on Information Security and Privacy*, pp. 274–291, Springer-Verlag, Sydney, Australia, July 2001. volume 2119, Lecture Notes in Computer Science.

[ML93] M. Löwe. "Algebraic approach to single-pushout graph transformation". *Theoretical Computer Science*, Vol. 109, pp. 181–224, 1993.

[MR99] P. McDaniel and A. Rubin. "A Response to "Can We Eliminate Certificate Revocation Lists?"". Technical Report 99.8.1, AT&T Labs, Florham Park, NJ, August 1999.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NL00] N. Li. *Delegation Logic: A Logic-based Approach to Distributed Authorization*. PhD thesis, New York University, September 2000. Chapter 4: A Nonmonotonic Delegation Logic.

[OpenPGP05] "An Open Specification for Pretty Good Privacy (openpgp)". `http://www.ietf.org/html.charters/openpgp-charter.html`, 2005.

[PKIX05] "The PKIX Working Group". `http://www.ietf.org/html.charters/pkix-charter.html`, 2005.

[PS00] P. Samarati, Ed. *Proceedings of the 7th ACM conference on Computer and Communications Security*, Athens, Greece, November 2000.

[RH98] R. Hirschfeld, Ed. *Financial Cryptography: Second International Conference, FC'98*. Springer-Verlag, February 1998. volume 1465, Lecture Notes in Computer Science.

[RLR98] R. L. Rivest. "Can We Eliminate Certificate Revocation Lists?". In: R. Hirschfeld, Ed., *Financial Cryptography: Second International Conference, FC'98*, pp. 178–183, Springer-

Verlag, February 1998. volume 1465, Lecture Notes in Computer Science.

[RS97] J. Rekers and A. Schürr. "Defining and Parsing Visual Languages with Layered Graph Grammars". *Journal of Visual Languages and Computing*, Vol. 8, No. 1, pp. 27–55, February 1997.

[RT99] M. Rudolf and G. Taentzer. "Introduction to the Language Concepts of AGG". `http://tfs.cs.tu-berlin.de/agg/intro.ps.zip`, 1999.

[SGS95] S. G. Stubblebine. "Recent-Secure Authentication: Enforcing Revocation in Distributed Systems". In: *Proceedings 1995 IEEE Symposium on Research in Security and Privacy*, pp. 224–234, May 1995.

[SW96] S. G. Stubblebine and R. N. Wright. "An Authentication Logic Supporting Synchronization, Revocation, and Recency". In: *Proceedings of the 3rd ACM Conference on Computer and Communications Security*, pp. 95–105, New Delhi, India, March 1996.

[TA98] T. Aura. "On the Structure of Delegation Networks". In: *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, Rockport, MA, June 1998.

[UM96] U. Maurer. "Modelling a Public-Key Infrastructure". In: E. Bertino *et al.*, Eds., *Proceedings of the 4th European Symposium on Research in Computer Security*, pp. 325–350, Springer-Verlag, September 1996. volume 1146, Lecture Notes in Computer Science.

[WLM00] R. N. Wright, P. D. Lincoln, and J. K. Millen. "Efficient Fault-Tolerant Certificate Revocation". In: P. Samarati, Ed., *Proceedings of the 7th ACM conference on Computer and Communications Security*, pp. 19–24, Athens, Greece, November 2000.

[WLM01] R. N. Wright, P. D. Lincoln, and J. K. Millen. "Depender graphs: A method of fault-tolerant certificate distribution". *Journal of Computer Security*, Vol. 9, No. 4, pp. 323–338, 2001. Erratum published in volume 10, number 3.

[X50004] "Information technology – Open Systems Interconnection – The Directory: Overview of concepts, models and services". 2004. ISO/IEC 9594-1:2001.

[X50900] "ITU-T Recommendation X.509 and ISO/IEC standard 9594-8:2001". `http://www.itu.int/ITU-T/asn1/database/itu-t/x/x509/2000/index.html`, March 2000.

Defence date February 17, 2006 **Publishing date (Electronic version)** January 26, 2006	**Department and Division** Department of Electrical Engineering Division of Information Theory

Language ☐ English ☐ Other (specify below) ______________	**Rapport category** ☒ Licentiate thesis ☐ Degree thesis ☐ Thesis C-level ☐ Thesis D-level ☐ Other (specify below) ______________	**ISBN:** 91-85457-84-1 **ISRN:** LIU-TEC-LIC 2006:1 **Title of series** Linköping Studies in Science and Technology. Thesis **Series number/ISSN** Thesis 1216 / ISSN: 02807971

URL, Electronic version
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-5477

Title

Understanding Certificate Revocation

Author(s)
Åsa Hagström

Abstract
Correct certificate revocation practices are essential to each public-key infrastructure. While there exist a number of protocols to achieve revocation in PKI systems, there has been very little work on the theory behind it: Which different types of revocation can be identified? What is the intended effect of a specific revocation type to the knowledge base of an entity?

As a first step towards a methodology for the development of reliable models, we present a graph-based formalism for specification and reasoning about the distribution and revocation of public keys and certificates. The model is an abstract generalization of existing PKIs and distributed in nature; each entity can issue certificates for public keys that they have confidence in, and distribute or revoke these to and from other entities.

Each entity has its own public-key base and can derive new knowledge by combining this knowledge with certificates signed with known keys. Each statement that is deduced or quoted within the system derives its support from original knowledge formed outside the system. When such original knowledge is removed, all statements that depended upon it are removed as well. Cyclic support is avoided through the use of support sets.

We define different revocation reasons and show how they can be modelled as specific actions. Revocation by removal, by inactivation, and by negation are all included. By policy, negative statements are the strongest, and positive are the weakest. Collisions are avoided by removing the weaker statement and, when necessary, its support.

Graph transformation rules are the chosen formalism. Rules are either interactive changes that can be applied by entities, or automatically applied deductions that keep the system sound and complete after the application of an interactive rule.

We show that the proposed model is sound and complete with respect to our definition of a valid state.

Keywords
Revocation, Digital certificates, Graph transformations, Distributed systems, PKI