Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups

By

David Lee [Chaum]

A.B. (University of California, San Diego) 1977
M.S. (University of California) 1979

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: Bernard Mont-Reynaud April 4, 1982
Chairman Date

Eugene L. Lawler April 4, 1982

J. Merrill Packer May 7, 1982

DEGREE CONFERRED June, 1982

Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups

Abstract

A number of organizations who do not trust one another can build and maintain a highly-secured computer system that they can all trust (if they can agree on a workable design). A variety of examples from both the public and private sector illustrate the need for these systems. Cryptographic techniques make such systems practical, by allowing stored and communicated data to be protected while only a small mechanism, called a vault, need be physically secured. Once a vault has been inspected and sealed, any attempt to open it will cause it to destroy its own information content, rendering the attack useless. A decision by a group of trustees can allow such a vault—or even a physically destroyed vault—to be re-established safely.

Networks of vaults can allow reliable operation even in the face of communication channel and vault failures. Networks also have several security advantages over single vault systems: (1) information that is no longer needed can be permanently destroyed, (2) comprehensive records of security relevant actions by the trustees can be maintained, and (3) abuse of the trustees' power requires advance notice. Algorithms which implement such a network are presented in a specially adapted formal specification language; examples of the algorithms' use are given; analysis of communication, memory and time requirements are presented; and security and reliability properties are proved.

Each of some mutually suspicious groups can supply part of a vault, in such a way that each group need only trust its part in order to be able to trust the entire vault. Another approach to construction is based on public selection of a system's component parts at random from a large store of equivalent parts. The practicality and ramifications of the ideas presented are also considered.

Table of Contents

Acknowledgements

The time spent discussing the material in this dissertation with Bernard Mon-Reyaud was extremely enjoyable, and lead to the development of an excellent friendship. Similarly, the time spent with Eugene Lawler, resulting from his participation in the thesis process was also quite enjoyable, and included many pleasant sessions during which, among other things, I received his counsel on the academic life in general and also on quite a range of particulars. Carlo Séquin and I had many stimulating discussions, mostly before and as a result of my masters thesis work, though he did contribute to the present work in a variety of ways, including participation in the qualifying exam. Bob Fabry and I worked together for well over a year, and his support during that time exceptional. Whit Diffie has contributed to the present work through numerous lengthy and extremely valuable discussions.

A few of my peers at Berkeley also contributed in no small way—interacting with them fulfilled ones expectations about how such relationships make gradschool: David Chin, Jim Demmel, Robert Hyrele, Peter Kessler, Keith Sklower, and Tim Winkler. This work started one summer, during which I spent a great deal of time with Doug Cooper. He helped me overcome a fear of writing. Manuel Blum was also around a lot that summer, and he and I talked. He maintained that one should never try to predict the effects of ones actions on society. It was the rejection of this principle which lead to the present work.

Chapter I

Introduction

§1 Problem Statement & Motivation

This section defines the increasingly important problem of providing computer systems that can be trusted by groups who don't necessarily trust one another. Example applications motivate the need for solutions and illustrate the nature of the solutions proposed.

Concern over the trustworthiness of computer systems is growing as the use of computers becomes more pervasive. It is not enough that the organization maintaining a computer system trusts it; many individuals and organizations may need to trust a particular computer system.

For example, consider a computer that maintains the checking account balances of a bank. The bank is concerned, among other things, about possible loss of balance records. The Federal Reserve Bank must know the total of these balances, to ensure that the legally required percentage of the balances is on deposit with it. The Internal Revenue Service requires the ability to check the balance of an individual's account. Individuals, or a consumer organization acting on their behalf, may wish to ensure that disclosures are made known to those

involved, and that inquiries can never be made on information that is more than a few years old.

There are many other similar applications of computers which involve private sector records related to consumers, such as those arising from credit, insurance, health care, and employment relationships. Public sector record keeping, in such areas as tax, social security, education, and military service are also quite similar.

Another class of applications involves information about public or private sector organizations as opposed to information about individuals. For example, various international agencies, such as the International Atomic Energy Agency, must be able to ensure the secrecy of the information they receive from their member nations. Numerous industry organizations develop statistics from confidential information submitted to them by their member corporations. Brokers and other middlemen in the mailing list industry must be able to a ensure the confidentiality of the lists they receive from a variety of list compiling organizations for purposes of removal of duplications or various kinds of prescreening.

All of these applications involve one group who owns or controls the computer system, and who is particularly concerned with reliably maintaining the operation of the system and with ensuring the survival of the data maintained by the system—they will be called the "trustees." A second group or set of groups are primarily concerned about the confidentiality of the data which relates to them that is available to the system. There may be a third group or set of groups, which may overlap with the first and second groups, who are concerned about the correctness of the operation of the system.

Of course, many applications of computer systems used solely within large organizations have a similar flavor, because such organizations are often composed of groups or individuals with conflicting interests.

§2 Overview & Chapter Summaries

The basic idea of the proposed systems is introduced and the organization of the thesis is presented as a guide to the reader.

This thesis offers a system design and feasibility argument for computer systems which can be established, maintained and trusted by mutually suspicious groups. Such systems can be used to meet the requirements of applications like those mentioned in the previous section, if a workable design can be agreed on by the participants. The cryptographic techniques which form the basis of the approach are introduced in the next chapter, Chapter II. They make such systems practical by reducing the mechanism upon which reliability and security depend. This mechanism—the processor and its high-speed store—will be called a *vault*. Vaults will be constructed in a way that can be verified by all the participants, or by any interested party, and then they will be physically secured, such as by being shielded within a small safe-like container.

In addition to introducing the cryptographic techniques, and presenting the relationship of the present work to the literature, Chapter II also surveys the varied literature which lends support to the practicality of the ideas presented: applications of cryptography; design and verification of security properties; securing apparatus from tampering and probing; and survivability of equipment, data and communication. Chapter III abstracts from the techniques of Chapter II the assumptions which form the basis of the proofs contained in a later chapter. At the same time, Chapter II also presents some important underlying assumptions which, although they do not enter directly into the proofs, influence the nature of the proposed systems. Chapter IV introduces a system based on a single vault. This serves the dual purpose of introducing a number of concepts used in the proposed multiple vault systems, and pointing out a number of shortcomings of single vault systems which are solved by the systems to be proposed.

The algorithms which define the operation of the multiple vault systems to be proposed are presented in Chapter V, using a specially adapted formal specification language. Then Chapter VI provides an example of the use of the algorithms, which demonstrates how a multiple vault system can be established. Proofs of various security and reliability properties are presented in Chapter VII, which make use of the assumptions of Chapter III. Analysis of the performance issues of space, communication, and time requirements of systems based on the algorithms of Chapter V is presented in Chapter VIII. Chapter IX presents techniques for constructing and placing into operation a secured vault, while maintaining the trust of potentially mutually suspicious groups. The final chapter, Chapter X, briefly considers work remaining and the implications of the present work.

Before delving into the supporting literature, however, it is important to indicate some of the unique contributions of the present work.

§3 What's So New About All This?

Suggested are the novelty and advantages of the present work over other work known to the author.

This thesis addresses the problem of establishing and maintaining computer systems that can be trusted by those who don't necessarily trust one another. This particular formulation of the problem is believed to be a contribution in its own right. In addition, the present work combines an unusually wide diversity of security technologies. The techniques presented for allowing construction of apparatus which can be trusted by mutually suspicious groups also appear to be new.

The detailed algorithms presented are the result of several major iterations, and are believed to take into account most of the important issues. The use of cryptography is central to many of the algorithms and is quite a bit more

complex than that reported elsewhere. This motivated substantial extension of a previously defined specification language in order to integrate a variety of cryptographic techniques into the type-checking and parameter-passing mechanisms in a convenient way. Also, a new general problem for computer network security, "the covert partitioning problem," is introduced along with algorithms which provide a solution and proofs of their correctness.

Chapter II

Survey of the Literature

Considered is some of the literature which lends support to the feasibility argument of the present work, and some related work.

This thesis puts forward a proposal for a new kind of highly secure computer system. The technologies upon which these systems must be based are quite diverse and cut across some traditional boundaries. Nevertheless, an attempt will be made to indicate the feasibility of the proposed systems by pointing to relevant surveys or directly into the literature.

§1 Cryptographic Algorithms

The various types of cryptographic algorithms used in the present work are discussed with reference to the relevant literature.

Information is encrypted to allow it to pass safely through a potentially hostile environment.

Conventional Cryptography

Secrecy. Traditionally, concern has centered on providing the confidentiality of message content. Consequently, cryptographic techniques were devised to make it very difficult (in some cases impossible) to transform encrypted information back to its unencrypted form without possession of a secret piece of information, called a *key*. Two correspondents who were the sole possessors of a key could use it to maintain the *secrecy* of the message content of their correspondences. Note that the cryptographic algorithms themselves are assumed to be public knowledge; only the key need be kept secret.

Ultimately, all cryptographic algorithms can be thought of as transforming symbols into other symbols. With a Captain Midnight decoder badge, the badge is the key, and letters are mapped into other letters. The un-breakable Vernam cipher maps only single bits into other bits, by adding each bit modulo two with a different key bit [Kahn 67]. On the other extreme, block cryptographic algorithms map large strings of bits, called blocks, into other blocks. The National Data Encryption Standard, for example, maps 64 bit blocks into 64 bit blocks, using a 56 bit key [NBS 77]. Many blocks can be "chained" together during encryption, effectively forming a single large block [Feistel 70].

Authentication. The present work assumes the use of block schemes, like the Data Encryption Standard, which make it very difficult to modify part of an encrypted block of information without causing drastic changes to the entire decrypted block. A large serial number can be appended to a block before encryption; its presence after decryption provides *authentication* of the block as a valid block that has not been altered. In such systems, it becomes extremely difficult for someone without a key to create a block that will contain a desired serial number when it is decrypted by a keyholder. Two communicants with a common key can converse using encrypted blocks of data, checking

the serial number of each received block to ensure that it has arrived in the proper sequence, and to ensure that it has not been altered [Feistel, Notz and Smith 75].

Public Key Cryptography

The cryptographic techniques considered so far have the unfortunate property that a common key must be distributed to the communicants, while it is kept secret from everyone else. In contrast, consider a fundamentally different sort of cryptographic algorithm independently proposed by Diffie and Hellman [76], and Merkle [78]. To use these algorithms, each participant creates a ***private key***, that is never revealed to anyone else. Only a suitably related ***public key*** is made known to everyone. Here we will be concerned with public key cryptographic algorithms (like that of Rivest, Shamir and Adleman [78]) where the two keys are inverses of one another, in the sense that a block encrypted with one can be decrypted only with the other.

Sealing. Public key cryptography can be used to provide the secrecy of message content. A confidential message can safely be sent if it is first ***sealed***, an operation which includes encryption with the recipient's public key. Only the intended recipient can decrypt the received message—because the corresponding private key must be used to decrypt it. A large random number is joined to the message during sealing, to counter two potential threats: (1) if the same message is sent more than once, such a message will be revealed as such to an eavesdropper; (2) an eavesdropper's guess of the message could be verified by encrypting the guess with the public key and then checking if the resulting bits are identical to the sealed message.

Signing. Authentication in public key cryptosystems is much more useful than that provided by conventional cryptography, because only a public key is

needed to authenticate a message, and hence anyone, not just the holder of a secret key, can check the authenticity of messages. Someone *signs* a message by encrypting it with their own private key. If a serial number of some agreed upon structure, such as all zeros for example, is joined to the message during signing, then its presence after decryption with the corresponding public key authenticates the signature.

Compression Functions

The so called "one-way" functions were introduced by Purdy [74] as part of the now familiar method of protecting passwords stored in computer systems. The one-way function and the image of all the passwords under the function are publicly readable, but they must be protected from alteration. Thus, the ideal one-way function is easily computed, but the inverse is computationally infeasible.

For the present work, a *compression function* will be a special kind of one-way function which maps an arbitrarily large domain into a fixed range, but which is practically impossible to invert. Such functions are quite handy since they in effect allow a relatively small number of signed bits to authenticate a large number of bits. Similar concepts have been described by various authors. (see Feistel [70] or Needham and Schroeder [78] for example.)

Key Generation

The automated generation of true physical random numbers has received some attention in the literature (see Knuth [7] for example). Sampling the noise generated by specially fabricated noise diodes seems to be an excellent source of raw bits (thermal noise and radioactive decay also seem good, but more cumbersome), which must then be corrected for bias in the detector.

Techniques for perfect correction of independent events with a fixed-bias detector are widely known. (Notice, however, that detector drift and physicl dependencies in the source contribute to less than perfectly independent raw bits.) The simplest such technique takes as input successive pairs of independent bits and outputs say a 1 bit for pairs of the form 1 0, outputs a 0 bit for pairs of the form 0 1, and produces no output for the other possible pairs 1 1 and 0 0 [Von Neuman 51; Gill 72]. It is also possible to combine many random numbers of some less than optimal entropy to produce a single number of increased entropy, such as by adding many numbers bit-wise modulo-two.

While details are beyond the scope of the present work, it is important to notice that many cryptographic algorithms may be quite weak for some choices of key. Care must be taken to determine if a candidate key is such a weak key and to randomly create another candidate in such a case.

§2 Applications of Cryptography

Discussed are some of the relatively few publications which assume good cryptographic algorithms and go on to consider applications.

Many kinds of security rely on the secrecy of their techniques. In contrast, much of the open literature on cryptography owes its existence to the premise that such secrecy may not be necessary or even desirable with cryptographic techniques. Shannon [49] assumes that the cryptographic algorithm is known to the "enemy" and only the key is secret. Kerckhoffs [1883] made a similar assumption. Baran [64] provides convincing arguments for making public the details of what he calls "cryptographic design" which includes the "hardware details".

There has been much work that considers the use of encryption for communications security and data security. The remainder of this section mentions some of the more relevant work in these areas. Work with a heavy emphasis on

the cryptographic algorithms themselves has been omitted, however, since this thesis is not concerned with particular cryptographic algorithms.

Communications security

Protocols that provide secrecy and authentication of communication between two devices using conventional cryptography are relatively straightforward and have been touched on by many authors, among them are Feistel, Notz, and Smith [75] and Kent [76]. Public key protocols for this kind of communication are similar to those based on conventional cryptography [Needham and Schroeder 78].

Key distribution. With conventional cryptography, the channel used to originally transmit the key from one participant to the other must provide both secrecy and authentication. Also, $O(n^2)$ keys can be required when n participants wish to converse amongst themselves using conventional cryptography. Heinrich and Kaufman [76] and Branstad [75] described an approach to distributing these keys that uses a central trusted device. (The techniques of the present work would be ideal if such an approach were to be used in an application with mutually suspicious participants.) Needham and Schroeder [78] describe both a centralized scheme and one in which the participants each use a trusted local device, all local devices having cryptographically secured communication amongst themselves. Diffie and Hellman [76] describe a scheme (devised with the collaboration of Lamport) which can only be corrupted by compromise of all of some fixed set of trusted devices.

The key distribution problem was at least part of the impetus for the two independent proposals of public key cryptography (Merkel [78] and Diffie and Hellman [76]). Only $O(n)$ keys are required by systems of the kind proposed by Diffie and Hellman. The key distribution problem is further simplified because

neither kind of system requires keys to be kept secret during distribution—only their authenticity must be ensured.

Traffic Analysis. The problem of keeping confidential who converses with whom, when and how much they converse, will become increasingly important with the growth of electronic mail. The problem of keeping an adversary from learning anything about the timing, amount or routing of messages in a communication system has been called the "traffic analysis problem." Baran [64] has solved the traffic analysis problem for networks using conventional cryptography, but his approach requires each participant to trust a common authority. In contrast, a system based on public key cryptography [Chaum 81], can be compromised only by subversion or conspiracy of all of a set of authorities. In the limiting case, each participant can be an authority.

The last approach allows one correspondent to remain anonymous to a second, while allowing the second to respond via an untraceable return address. This permits rosters of untraceable digital pseudonyms to be formed from selected applications. Applicants retain the exclusive ability to make digital signatures corresponding to their pseudonyms. Elections in which any interested party can verify that the ballots have been properly counted are possible if anonymously mailed ballots are signed with pseudonyms from a roster of registered voters. Another use allows an individual to correspond with a record-keeping organization under a unique pseudonym which appears in a roster of acceptable clients.

Data Security

Conventional cryptography has received some consideration as a technique for protecting stored information. The use of encryption to protect objects within operating systems, first suggested by Peterson and Turn [67], suffers

from the problem of key management. One might argue that whatever techniques were applied to protect the keys, might have been applied to the data itself, thus eliminating the need for encryption. But advantage can be taken of the small, fixed-size of the keys.

The use of cryptographic techniques to protect data stored in a potentially hostile environment are relevant to the present work. There are three important considerations for protecting stored data, each corresponding to one of the issues of secrecy, authentication, and traffic analysis in the context of communication. First, if the same data is stored more than once under the same key, then some non-repeating data, such as the random serial number used in sealing, must be included in the data lest the repetition be revealed. Second, it may not be sufficient to be able to authenticate the memory location associated with a page received from storage if data has been stored at that location more than once; a solution to this, the "most recentness" problem, must be provided so that the page can be authenticated as the last copy written. (Solutions to this problem which also solve the first problem are presented in the work of Bayer and Metzger [76] mentioned below.) Finally, the pattern of read and write accesses must be considered as a possible source of information to an adversary. A most general solution to this last problem, which makes no assumptions about the application program, might be to alternately read every stored location ever written and then to perform a fixed number of writes. Clearly this is not an attractive solution, and much more reasonable solutions, possibly including the introduction of some bogus requests, can be developed by careful design of the application program.

An interesting technique has been developed for encrypting information which is divided into pages. A different key is used to encrypt each page. The key used for a particular page is produced by encrypting the address of the page using a master key. Mapped addresses (so that addresses can be changed

for new versions of a page) and physical addresses are considered by Bayer and Metzger [76]. Content addresses have been dealt with by Gudes, Koch, and Stahl [70]; and by Flynn and Campasano [78].

Some simple systems have actually been built that encrypt data at a secure site before transmitting it to an un-secured data base management system [Notz and Smith 72; Carson, Summers and Welch 77]. The terminals or their users are presumably the only holders of the keys so that only they can access the data.

§3 Partial Key Techniques

Various solutions to the problem of dividing a key, or other secret information, between individuals or other entities are presented.

Feistel [70] describes schemes in which a cryptographic key is divided into n parts, each part is given to a different person, and the original key can be re-created by combining all n parts. These schemes use random bits for each part except the last, which is chosen so that the desired key is the bit-wise modulo-two sum of this last part and the rest of the parts. A disadvantage of such schemes is that if just one part is lost, then the original key can not be re-created.

The technical report on which this thesis is based (Chaum [79]) introduced a scheme for dividing a key into parts, called *partial keys*, in which some selected subsets of the partial keys are sufficient to re-create the original key. The approach used was based on multiple encryption. Independently, and at about the same time, Blakley [79] and Shamir [79] published more elegant schemes which do not have the inherent flexibility of the multiple encryption schemes, but can use less space and run faster for large n when the required sets are all possible sets with cardinality greater than some fixed number. These techniques are unbreakable as is the Vernam cipher mentioned earlier,

and the Vernam cipher has even been called a degenerate case of these techniques [Blakley 80]. Further work by Azmuth and Bloom [80] includes means for determining which if any partial keys submitted for a re-creation are bogus.

§4 Computer Security

The field of computer security is divided into four areas, and each is dealt with in a separate subsection.

Computer security is the topic of several journals, several annual conferences, dozens of books, thousands of articles in the technical literature, and many more pieces in the popular press. It is far beyond the scope of the present work to try to survey this vast literature.

For the purposes of this section, the field of computer security is divided into four broad groups of concerns:

(1) issues related to personnel and their access to facilities;

(2) design of desired security properties;

(3) verification of implementation of the desired security properties;

(4) physical security of equipment against probing and modification.

Survivability issues are covered in the next section.

Personnel

Discussion of personnel issues are liberally sprinkled throughout the computer security literature, particularly that aimed at the practitioner. From the technical point of view, the major issues with respect to personnel are how to reduce the exposure to personnel, and then how to force conspiracies of persons for what exposure remains. Essentially two ways to force conspiracy are used. The most desirable mechanisms are those which can force equally knowledge-

able persons to conspire. For example, the so called "two man rule," used for control of nuclear weapons, may require that two keys located at substantial distance from one another be turned simultaneously. A somewhat less appealing but much more widely used approach is to attempt to limit the knowledge of individuals to such narrow aspects of a system that they must conspire with others in order to have the knowledge and skills required to compromise the system (see [FDIC 77] for example). Since the present proposal uses equipment which is essentially inaccessible to personnel, and techniques which are a generalization and extension of the two man rule, many of the personnel issues are not particularly relevant.

Other questions raised in this literature include: How can trustworthy personnel be selected? What sort of "access control" mechanisms are appropriate for controlling the movements of people into and within a facility? What is the best way to motivate compliance with security relevant rules? and How can the user interface of the security mechanism best be designed so as not to encourage bypassing by the user?

In any system in which personnel must be trusted, the possibility always exists of influence by positive means such as bribery, negative means such as blackmail, and the combination. Also, one can never be sure that a person's behavior will remain uniform. For example, stress in personal life, breakdown, suggestion and drugs can cause substantial changes in behavior.

Protection

In some dedicated applications, such as some of those mentioned earlier for which the present work may be particularly well suited, answers to the question Who can make what kind of accesses to what data? may be quite obvious and simple. In more general purpose systems, such as operating systems and data-

base management systems, it may be difficult to decide on a way to describe the kinds of accesses allowed. There may be various design objectives, such as, closeness of fit to anticipated application requirements, ease of user understanding, implementation efficiency, appropriate default rights, congruence with user motivation, and convenience of use.

For operating systems, the proposed access control models are often divided between the "access control matrix" approaches [Lampson 74], and the "information flow" approaches [Denning 76]. In the access control model, a matrix contains the type of access allowed by each of a set of subjects to each of a set of objects. Data flow is a generalization of the U.S. classification scheme, which was based on the British scheme, where information is allowed to flow up to higher classifications but not down to lower classifications. Recently, Stoughton [81] has proposed a synthesis of the two approaches. In database management systems, the protection structures proposed may be divided between the access control style and the value dependent. An interesting approach called query modification has been suggested [Stonebraker 75], in which additional restrictions are automatically appended to each query before it can be processed.

The general case is further complicated because provisions must be made which allow access rights to be changed and even for the rights related to who can change access rights to themselves be changed. Much theoretical work, such as that of Harrison, Ruzzo & Ullman [76], demonstrates that it may not be practical to determine who ultimately may access what, even with rather limited kinds of transfer rights.

In general, when preventive means are not available, it may still be possible to preserve a record which reveals abuses. Thus, various "logging" or "audit trail" techniques have been proposed, such as those of Weissman [69].

Verification

In the present context, *verification* is intended to mean the process of developing certainty that some formally described mechanism has some desired properties; the term *certification* is used here to mean that some physical mechanism "conforms" to the formal description. This subsection points into the relevent literature on verification; little has been found in the literature on certification (but see Weisman [69]), a topic which is covered in Chapter IX. The field of program verification was given a formal foundation by Floyd [67]. He defined a program to be "partially correct" (with respect to some input and output assertions) if the truth of the input assertions before program execution guarantees the truth of the output assertions after program termination. (A "totally correct" program was a partially correct program whose execution is guaranteed to terminate.) He gave a method based on inductive assertions for determining partial correctness of programs. Proof techniques for parallel programs have also appeared (see Owicki and Gries [76] for example). Proving properties about cryptographic protocols is also receiving attention (see Dolev and Yao [81] for example).

A variety of automated specification and verification systems have been developed and are extensively used for security work (see Cheheyl et al [81] for a recent survey). In such systems, formal specification languages are used to define the intended function of a module, while omitting as much implementation detail as possible (see Rammamoorthy and So [81] for a survey). For example, the HDM (Hierarchical Design Methodology) [Robinson and Levit 77; Levitt, Robinson and Silverberg 79] uses the specification language presented by Parnas [72] to describe systems as a hierarchy of abstract machines. (The Parnas specification language is extended in Chapter V and used to present the algorithms proposed here.) Global and local security properties of programs executing on multiple processors, and employing cryptographic techniques, of much

the same order of complexity as those algorithms presented in Chapter V have recently been verified [Good et al 82].

§5 Physical Security

The little open literature on protecting equipment from probing and modification is considered.

Shielding techniques for protecting mechanisms against analysis of their radiated signal energy, or probing by externally supplied energy, seem to be rather well understood, and are covered by the classified TEMPEST specifications.

Tamper-safing systems can be divided between those which merely indicate tampering to an inspector, and those systems which can detect tampering and can respond by, for example, destroying some secret information. In some cases it may be desirable to augment a *tamper-responding* system with *tamper-indicating* techniques and periodic inspections. (See the next chapter for more on combinations.) There is a small amount of unclassified literature on tamper-indicating techniques [Poli 78], but almost nothing on high level tamper-responding techniques—but see Chaum [82].

One approach to the problems of TEMPEST and tamper-safing includes placing apparatus to be protected in relatively inaccessible locations. For example, satellites or satellite platforms may provide an ideal location because it becomes very difficult to surreptitiously compromise equipment in such a visible and inaccessible location, or to get close enough to obtain an acceptable signal to noise ratio from even moderately well shielded equipment. (Such locations may also be quite attractive because of the kinds of communication channel typically provided by satellites, as mentioned later.)

Another location which has great potential, and has actually been used for protecting apparatus (see Sandia [81] for example), is the bottom of well holes

in rock formations. Seismic sensors do a good job of detecting attempts to come even moderately close to the protected apparatus.

Installations in office building environments are also possible. While it is beyond the scope of the present work to discuss the various possibilities for solving these problems in less remote locations, it may suffice to point out that tamper safing and shielding have obvious importance in intelligence and military systems, and one can safely assume that these problems have been adequately solved for these applications. Thus, it appears that the physical security requirements of the applications considered earlier are quite reasonable.

§6 Survivability

This section surveys the issues in survivable systems, which include barriers or hardening, redundant communication, redundant storage, and reliable mechanisms.

As in the previous section, the requirements of the kinds of applications considered will appear quite practical based on the following discussion.

Barriers

The problem of providing substantial resource requirements and delays to would be penetrators has been referred to as the *barriers* problem in the nuclear safeguards literature [Sandia 78]. Acceptable barriers for some applications can be provided by concrete and steel structures, but more sophisticated barriers are constantly under development by the manufacturers of better safes and vaults. Such developments are rarely published and are only alluded to in sales literature. Unfortunately, the so called "shaped charge" can almost instantly penetrate any barrier of reasonable thickness. But, quite satisfactory barriers can be provided by placing equipment to be protected in inaccessible

locations, such as the well holes described in the previous section.

Reliable Equipment

Largely because of developments in space and aviation, computer systems and related equipment have been developed which use redundant mechanism to achieve extremely high reliability. (See Randell et al [78] for a relatively recent survey.) Some of these advances are already enjoying widespread use in earth-bound business transaction processing systems, and are likely to become increasingly more widespread because of trends such as decreasing hardware costs and increased dependency on real-time systems. Thus, for the sorts of applications the present work is directed at, highly reliable systems may be rather common.

Survivability of Data

One very nice thing about safely encrypted data is that a proliferation of copies does not pose any additional threat to security, but it has great potential for increased survivability. Multiple copies of encrypted data can exist at a variety of sites, some of which may be hardened. Also, when broadcast style communication channels are used, locations which are maintaining copies of data may not even be known to the issuer of the data, and might therefore be extremely difficult for an adversary to even detect. Today, several companies provide secure data storage sites for magnetic recording media. Some of the facilities are located deep within mountains while another is in an abandoned telephone switching center which was hardened to withstand a several megaton blast.

Survivability of Communication

The ability to communicate in spite of an adversary is of obvious importance for military applications. The use of redundant and alternate channels is one standard approach to the problem [Frank & Frisch 70]. Other more effective approaches are under development and in use, however, they receive little coverage in the literature. One important approach is the use of cryptographically controlled "spread spectrum" radio techniques, which provide a broadcast signal which is nearly impossible to jam [Haakinson 78]. Also highly redundant error correcting codes can greatly increase the survivability of transmissions in a noisy environment.

§7 Related Work

A few extended citations give credit to some relevant earlier work.

It seems appropriate to include this section to put the present work in perspective with some proposals of others addressed at similar problems.

Feistel might be called the father of modern conventional public cryptography. His plan and motivation for non-military use of cryptography comes through in the first part of his introduction to "Cryptographic Coding for Data-Bank Privacy," which is excerpted below. This document remained classified "IBM CONFIDENTIAL" for a couple of years after it originally became a "Research Report" in 1970.

> A Data Bank is essentially a machine to machine communications network in which input terminals are connected to a centrally located computer, the physically secured CPU.
>
> The most outstanding feature of the kind of network structure we are talking about is that it must function reliably in a hostile environment. Secrecy in the usual sense, that is concealment of the meaning of the messages

> conveyed, would form the basic element of protection. This is required to insure the privacy of those forming the data bank community. But machine communications systems, in contrast to systems which can enlist the subtle filtering capabilities of the human brain are very sensitive to interference and deception. Without special protection computers are easily fooled and this can become an intolerable burden to a data bank operation if this remains unnoticed. Both accidental and intentionally designed errors must be detected with very large safety margins. A machine to machine communications network requires a properly secured method which assures the receiver that all incoming communications are of legitimate origin and uncorrupted. In military systems such methods are called authentication. We shall present a method called centralized verification. In contrast to military systems, where all participants have the same key, our system emphasizing individual privacy permits each individual member of the data bank to have his own private key. . . .
>
> The heart of our Data Bank Network is the so called Vault, which is properly secured physical location of the central data processing facility consisting of a time sharing CPU and appropriate storage or filing facilities.

Schroeder realized, early on, that many important applications of computer systems could involve groups with conflicting interests. His dissertation, "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," evolved out of work on MULTICS under Saltzer, at MIT, and also appeared as a Project MAC technical report in 1972. The following excerpts indicate the motivation and scope of his work.

> This thesis describes practical protection mechanisms that allow mutually suspicious subsystems to cooperate in a single computation and still be protected from one another. The mechanisms are based on the division of a computation into independent domains of access privilege, each of which may encapsulate a protected system. The central component of the mechanisms is a hardware processor that automatically enforces the access constraints associated with a multidomain computation implemented as a single execution point in a segmented virtual memory. . . .
>
> In this thesis interest is centered on protection mechanisms within computer systems. The viewpoint is that of a computer system designer who is intent upon providing efficient protection mechanisms applicable to a wide range of problems. Questions of privacy influence this effort to the

extent of implying criteria which must be met before such a computer system can be applied to those problems where privacy is an issue. The thesis, however, contains little explicit consideration of privacy.

To further define the scope of the thesis, consideration is limited to problems of hardware and software organization. While it is recognized that issues such as installation security, communication line security, hardware reliability, and correctness of hardware and software implementations of algorithms must be considered in order to achieve the secure environment required for useful application of protection mechanisms, these topics are beyond the scope of the thesis. . . .

Taken together, the hardware and software mechanisms described in this thesis constitute an existence proof of the feasibility of building protection mechanisms for a computer utility that allow multiple user-defined protected subsystems, mutually suspicious of one another, to cooperate in a single computation in an efficient and natural way.

Parker has provided the public with many amusing tales of crimes perpetrated by individuals against organizations maintaining computer systems. While the present work tends to be concerned with protecting individuals or groups from organizations maintaining computer systems, the solution envisaged by Parker in his 1976 copyright book, *Crime by Computer*, is quite instructive.

It must become clear to the business community, government, and finally the public that the safety of our economy and our society is growing increasingly dependent on the safe use of secure computers.

An ideal secure computer system including data communication capability would be one of proven design which could be run safe from compromise without human intervention. It would be served by computer operators who would be allowed only to perform tasks directed by and closely monitored by the computer. No maintenance by human beings would be allowed in its secure operational state. All failures short of being physically damaged from an external force would be failsafe, and a failure not automatically reparable or overcome would cause the system to shut down in an orderly, safe fashion and lock up all

data files in a separate, secure storage.

It might take four trusted executives, including a special government inspector, simultaineously to insert and turn keys in the system console locks to change the mode of operation from "secure" to "open." Then human access to modify and repair the system would be allowed. Before returning the system to secure state again, a team of auditors would go through an elaborate process of reproving and testing the secure state. Once the system is again declared secure, another group of four executives would simultaneously turn their keys in the console locks to make the system again operable in secure state.

Chapter III

Assumptions

This chapter is intended to make sufficient assumptions so that the proofs of Chapter VII can be completed. In addition, the fundamental assumptions which shape the proposed design are presented.

In the first two sections of this chapter, notation is presented for the cryptographic techniques introduced in the previous chapter, and this notation is then used to describe the properties desired of the techniques. Section three makes explicit the assumptions about certification used in the proofs of chapter VII. (Certification of vaults is covered in Chapter IX.) The last two sections of the chapter present the assumptions about physical security and organizational structure which shape the design of the proposed systems.

§1 Cryptologic

Defines exactly what a crypto-system is assumed to make intractable.

It will be assumed that the possibility of successful "forgery," "sealbreaking," or "de-compression" efforts, using feasible amounts of computation, is so small that it can safely be ignored.

Notation. Someone becomes a user of a public key cryptosystem by creating a pair of keys, K and K^{-1}, from a suitable randomly generated seed. The public key K is made known to the other users, or anyone else who cares to know it; the private key K^{-1} is never divulged. The encryption of X with key K will be denoted $K(X)$, and is just the image of X under the mapping implemented by the cryptographic algorithm using key K. The increased utility of these algorithms over conventional algorithms results because the two keys are inverses of each other, in the sense that $K^{-1}(K(X)) = K(K^{-1}(X)) = X$.

Forgery

A user signs some material X by prepending a large constant C (all zeros, e.g.) and then encrypting with its private key, denoted $K^{-1}(C,X) = Y$. Anyone can verify that Y has been signed by the the holder of K^{-1}, and determine the signed matter X, by forming $K(Y) = C,X$, and checking for C.

A digital signature is forged by someone who creates it without the appropriate private key K^{-1}. A potential forger is assumed to have the public key K and the ability to have some items of the forger's choice signed. A forgery attempt is considered successful if it yields some item Y that has not been signed using the private key but for which $K(Y) = C,X$, regardless of what X is. One forgery strategy is to choose values for Y at random, until one is found whose decryption with K yields something with a prefix of C.

An alternative attack that is of general utility requires only a public key. The corresponding private key can be found by using candidate private keys to decrypt an item encrypted with the public key, until one such decryption yields the original item.

Sealbreaking

The sealing of X with K, is denoted $K(R,X)$, where R is a random string.

A potential sealbreaker is assumed to have the public key K, a set of items of uniform size and another set containing the items of the first set in sealed form. A successful sealbreaker knows something about the correspondence between the elements of the two sets. One sealbreaking strategy is to guess the random information R that was used to seal one particular item from the unsealed set. Prepending the guess to the item and encrypting with the public key would yield an item from the set of sealed items only if the guess were correct. This would reveal a single correspondence.

De-Compression

A compression function F maps a large string of domain bits D into a roughly key-sized string of bits R. The adversary is assumed to have the function F, an element of the domain D of interest, and of course $R = F(D)$. The adversary is successful if a second element of the domain, D', can be produced such that $D' \neq D$ and $R = F(D')$.

§2 Partial Key Techniques

Defines what is expected of a partial key technique, and also makes significant assumptions about their use.

It is assumed that the possibility of someone not privy to the seed or sufficient partial keys being able to determine anything about the original key which was divided is so small it can safely be ignored. Also, knowledge of even chosen partial keys never gives any clue about the seed used.

The entities holding partial keys in this thesis will be assumed "equally capable". In other words, they will be a homogeneous set, any subset of cardinality greater than the threshold value established for the partial keys will be sufficient to reconstruct the original key. A similar homogeneity assumption will be made about other kinds of voting as well. These assumptions strongly flavor the approach presented in the following chapters. The possibility of other approaches is mentioned in Chapter X.

§3 Verification & Certification

Defines the requirements of verification and initial certification.

Assume that mutually suspicious groups can know that the plan for a vault has the desired properties and that the vault operates correctly according to the plan, as a result of some verification and certification procedures. Verification was discussed in the previous chapter; some new approaches to performing certification are the topic of Chapter IX.

§4 Physical Security & Survivability

Potential attacks on a vault are described and compared.

This section presents a list of possible attacks on a vault. The results of these attacks vary from total covert control of a vault by an attacker, to simple destruction of a vault. The following is a summary of the potential threats against a vault, roughly in decreasing order of difficulty:

(1) Surreptitious corruption—vault has been modified, and secret keys within vault may be known; the attack is not detectable by inspection; both tamper-indicating and tamper-responding mechanisms have been defeated.

(2) Detectable corruption—same as (1) but inspection will reveal at least attempted tampering; tamper-indicating mechanism has not been defeated.

(3) Compromise—secret information within the vault has become known to attacker, but the attack leaves no trace; attack may consist of probing, limited compromise of the tamper-safing mechanism, exploitation of weaknesses in the TEMPEST techniques employed, or possibly cryptanalysis.

(4) Covert isolation—node kept from communicating with anyone except attackers; node presumed dead to observers; may be a difficult attack where broadcast style communication channels are used.

(5) Overt isolation—communication with outside blocked; attack obvious to observers; e.g. jamming in a system with broadcast style communication.

(6) Destruction—vault is disabled.

§5 Organizational Structure

Defines the three tier organizational structure assumed for the most elaborate application of the proposed systems.

Chapter I mentioned the existence of one group in a computer application that is particularly concerned with reliability and survivability of the system. The systems design proposed in subsequent chapters further divides this group into three different bodies, called *trustees*. The analogs of these bodies in a corporation might be its officers, directors, and stockholders. The following table summarizes the functions and exposure to the three levels of trustees:

(1) trustee level 1—charged with day-to-day operations of the system, which include implementing a policy which balances survivability and performance, within the policy constraints formulated by the trustees at level 2; has no significant advantage in attacking security over anyone.

(2) Trustee level 2—charged with policy formation aspects of trusteeship, in which the trustees at level 2 must define how difficult it will be for them and also how difficult it will be for others to defeat the system, to decide which

new vaults will be used; will be able to compromise some security properties without any attack, but only after giving advance notice.

(3) Trustee level 3—charged with the ability to restore the whole system in the event of disaster; can perpetrate certain threats without any attack.

Chapter IV

Single Vault Systems

A simple single vault system is presented to introduce and illustrate some of the basic ideas of the proposed systems, and also to motivate and define the problems to be overcome by multiple vault systems.

When a certified vault is first constructed by the techniques presented in Chapter IX, a suitable public key and its inverse private key are chosen by a mechanism within the vault's protected interior, using a physically random process as discussed in Chapter II. The public key is then displayed outside the vault, on a special device certified for this purpose. As far as the world outside the vault is concerned, the possessor of the vault's private key is the vault: it can read sealed confidential messages sent to the vault, and it can make the vault's signature.

§1 Checkpoints & Restarts

Introduces the notions of encrypted checkpoints and the restarts they can allow trustees to perform.

What if Something Goes Wrong?

If a vault were totally destroyed, computation would be safely halted—no secret information would be revealed, and the vault would not have taken any improper action. Other conditions might require an equally safe halt to computation. If a tamper-responding system detects an attempt to penetrate the vault's protective enclosure, or a fail-safe mechanism determines that the vault's contents can no longer be counted on to operate correctly, then the information stored in the vault, including the vault's private key, must be erased.

This information will be encrypted in a special way, and saved outside the vault, so that a safe recovery can be provided. The encryption of the vault's contents, which includes its private key, is called a *checkpoint*, and is detailed below. At suitable intervals, checkpoints are formed, and then stored outside the vault. In some cases, there may be time to issue un-scheduled checkpoints before an emergency requires the vault's contents to be erased.

The primary consideration behind the design of an encryption method for checkpoints is that there exists a means to decrypt them, but only at the appropriate time and place. The decision that some newly sealed vault can, and should, be given the ability to decrypt a checkpoint is necessarily a human one. Assume, for now, that the decision is to be made by unanimous consent of a set of human trustees. Before a checkpoint is released by a vault, it is encrypted with a special key for this purpose. Conventional as opposed to public key cryptography can be used for this. This key used to encrypt checkpoints will be divided into partial keys, one key for each trustee.

Public key cryptography will be used to distribute these partial keys to the trustees in a secure manner. As part of the certification process, the vault is supplied with a public key issued by each trustee. Thus, the vault can ensure the confidentiality of the partial key it sends each trustee by sealing that partial

key using the trustee's public key. Each trustee now has two keys to keep secret: a private key used to unseal messages received, and a partial key that will be used in connection with decrypting checkpoints.

Restarts

A *restart* is the process by which a freshly sealed vault resumes the computation whose state has been saved in a checkpoint. After a replacement vault is certified and sealed, it forms a temporary public key and its inverse private key from a random seed, and then displays the temporary public key, as the permanent public key was displayed in the original start-up. Then the restarting vault receives partial keys from the trustees. A trustee provides the secrecy of its partial key while it is in transit to the vault by sealing it with the displayed temporary public key.

Having received and decrypted the partial keys, the computation within the replacement vault merges them to form the key originally used to encrypt checkpoints, and uses this to decrypt the checkpoint received. The replacement vault then bootstraps itself into the state saved in the checkpoint. Thus, the original public key found in the checkpoint is reinstated, and the computation within the replacing vault becomes an exact copy of the original computation. The restarted vault can then be safely brought back up to date by replaying all the messages sent it since the checkpoint was made.

§2 Limitations of Single Vault Systems

Several kinds of abuse of single vault systems by the trustees are described and solutions using multiple vault systems are sketched.

It is generally held that networks of computers may be better than a single centralized computer system in many applications, for such reasons as

improved performance, increased reliability, and decreased communication costs. The multiple vault systems to be presented in the following chapters may be preferred over single vault systems for similar reasons. In addition to the usual advantages, however, multiple vault systems offer solutions to many of the problems of single vault systems:

Destruction of Information. In a single vault system, the partial keys held by the trustees will always be sufficient to decrypt any previous checkpoint. Thus, a conspiracy of a sufficient subset of the trustees will have access to all information, no matter how old the information is. In a multiple vault network, however, the trustees will be forced to request certain partial keys from the network during a restart in order to obtain sufficient partial keys to decrypt a checkpoint. The network will change the keys used to form checkpoints, and the partial keys it maintains, in such a way that obsolete checkpoints can never be decrypted. (A conspiracy of trustees in a single vault system need never be able to forge a vault's signature, since a private key used by a vault only for making signatures need never be saved outside the vault.)

Comprehensive Record of Restarts. In a single vault system, a conspiring subset of the trustees can secretly combine their partial keys and obtain keys sufficient to allow them to decrypt checkpoints. In the multiple vault system, the trustees will have to request partial keys from the network to accomplish a restart, as mentioned above, and the network will be able to maintain a record guaranteed to include descriptions of all such requested restarts. Such a record is very useful because it can ensure that only certified vaults have decrypted checkpoints, and that they have done so only during certified restarts.

Advance Notice of Security-Relevant Changes. In a single vault system, the trustees can perform a restart using a vault which is certified but which contains an arbitrary change in the security-relevant aspects of the vault's

operation. For example, the new vault may give greater power of inspection or modification to the trustees. In multiple vault systems, the trustees can be required to give advance notice of security-relevant changes, such as the public keys of vaults to be added into the network and changes in parameters used by the network to protect itself from the trustees.

Chapter V

Multiple Vault Systems

Algorithms to be performed by a collection of vaults are defined using an extended formal specification language.

§1 Introduction to Algorithms

An overview of the algorithms proposed is presented which includes the relationship of this chapter to other chapters.

This chapter describes a collection of algorithms to be performed by a number of separate vaults, or *nodes*. Each node will perform essentially the same algorithms, but some of its own state may vary. The algorithms are organized as a set of a dozen and a half independently callable routines. A node will perform any one of these routines on request, if it is provided with the appropriate actual parameters. Typically, some of the actual parameters of a call will bear digital signatures formed by other nodes in the system and also by various trustees. If these signatures and the rest of the parameters prove acceptable to the called routine within a node, then the node may alter its state and/or produce some signed and possibly sealed output as a result of performing the called routine. Calls are handled one at a time by a node, so that once a node com-

pletes processing of one call, it begins waiting for the next call to be requested.

The nature of the algorithms and their use of cryptographic techniques ensure that: (1) the various security properties provided by the system can not be violated by any sequence of calls, and (2) the trustees can maintain the reliable operation of the network by performing suitable sequences of calls. Chapter VII argues these points; the present chapter uses a specification language to describe a practical version of the algorithms.

Among other things, the algorithms must provide a kind of synchronization and agreement among nodes about allowing new nodes into the network, removing nodes from the network, and the status of nodes once in the network. The routines will be called *O*-functions (for *O*peration function) since they are an extension of the *O*-functions of the Parnas specification language [Parnas 72], as mentioned in Chapter II. Figure 1 shows seven of the major *O*-functions. These *O*-functions can change the membership of the network and the status of nodes within the network. For example, the *CERTIFY* *O*-function can bring a new node into the network, leaving the new node in the "initiate" state. Similarly, *REMOVE_NODES* can take a node in the "participated", "veteran" or initiate states out of the network. These and the other *O*-functions will be described in detail in sections 6 and 7.

Section 2 introduces the basic types, primitives and constants of the specification language. Section 3 and 4 define the state of nodes as a collection of *V*-functions (for *V*alue function), which have been extended to include types not in the original Parnas notation. Section 5 defines the rather powerful parameter passing mechanism used both for input and output by the *O*-functions, which is an extension of the Parnas notation. Finally, as mentioned above, sections 6 and 7 present specifications of the *O*-functions themselves.

Figure 1. *O*-functions & Their Node State Transitions

Strictly speaking, a specification language is intended to define what a program is to do—and not how it is to do it. Nevertheless, it will be very convenient to apply the familiar terminology of programming languages to the specification language used here. The presentation of the specification language will also use a variety of type fonts and type sizes, roughly based on those used by Parnas [72]. Some symbols will appear in upper case, others in lower case, and a few others will combine the two. A summary of the typographic conventions is presented in Table 1.

primitives & constants
syntax-meta-symbols
pseudo-types
types
type-constructors, if then else & with
PARAMETER_NAMES & TEMPORARY_VARIABLES
V-FUNCTION_NAMES & O-FUNCTION_NAMES
AGGREGATE-FUNCTION_NAMES

Table 1. Typographic Conventions

§2 Simple Types, Primitives & Constants

The basic data types of the specification language and the elementary operations which can be performed on them are presented.

The specification language is strongly typed, although some primitive functions can have arguments of any type. Some primitive functions have no arguments, but those entities with fixed values are called constants.

Simple Types

Some of the simple types are those usually found in programming languages. Others are the keys, seeds, and parts of keys used by the cryptographic transformations. Yet others are simply enumerated types, *a la* Pascal, used as tags included in signed messages to indicate the kind of message. A special type is used to represent node names. Chapter VIII contains some discussion of straightforward representation schemes for instances of the simple types, and the constructed types of the next subsection, for purposes of analysis, but further consideration of implementation techniques is beyond the scope of this work.

A simple context free grammar will be used to illustrate the basic syntax of the specification language. The first production of the grammar is shown here:

elementary-type → **boolean** | **integer** | **time** | **node-id** |
seed | **public-key** | **private-key** | **partial-key** |
proposal-kind | **announcement-kind** | **action-kind** | **transfer-kind**

The following is a detailed definition of each of the elementary types:

boolean, integer The usual.

time The content of a clock or counter. Uniform units are used so that the difference of two times produces an **integer** which is proportional to the amount of time between the two times.

node-id A special type whose values are used to uniquely identify nodes and trustees, and whose values are never re-assigned.

seed A randomly selected value preferably from a space at least as large as the space of possible keys, which is returned by the primitive function *create-seed* and is used by the primitive functions *create-public*, *create-private*, and *form-partial*, to create keys and partial-keys.

public-key A public key that was created by a call to *create-public*. Generaly publicly available, and can be a parameter in calls to *seal* and *check-signature*.

private-key A private key that was created by a call to *create-private*. Generally kept secret by its creator, except may be transferred during a *RESTART*. Used in calls to *sign* and *unseal*.

partial-key A partial value of a **private-key** that is created by a call to *form-partial*. Sufficient quantities of these keys can be used by *merge-partials* to reconstruct the private key from which they were formed.

proposal-kind This is an enumerated type, *a la* Pascal, whose values are denoted by the constants: *propose-certify propose-set-minima* and *propose-remove*. They are used as inclusions in signed proposals of the corresponding names.

announcement-kind An enumerated type, whose values are used as inclusions in announcements of proposed actions of the corresponding names. The unique values are denoted by the constants: *certify*, *set-minima*, and *remove*.

action-kind Used as an inclusion in signed announcements of trustee level 1 actions. The unique values are denoted by the constants: *propose*, *cancel*, *apply*, *change-presents*, *restart*, *participate*, *create-keys*, and *change-keys*.

transfer-kind Used as an inclusion in signed output generated by an *O*-function and intended to be consumed by one or two different *O*-functions. The unique values are denoted by the constants: *RESTART_to_ASSUME_APPLICATION*, *PARTICIPATE_to_RECEIVE_NEW_PARTICIPANT*, *PARTICIPATE_to_NEW_PARTICIPANT_RECEIVE*, *CREATE_KEYS_to_ISSUE_NEW_PARTIALS&CHANGE_KEYS*, *CREATE_KEYS_to_NEW_PARTICIPANT_RECEIVE*, *ISSUE_NEW_PARTIALS_to_RECEIVE_NEW_PARTIALS*, *RESTART_to_ASSUME_APPLICATION*, *partials-received*, *proposal*, and *checkpoint*.

Constructed Types

The elementary types of the previous subsection may be combined into sets or tables. This is an extension of the original notation proposed by Parnas and further developed for HDM [Levi, Robinson and Silverberg 79], but resembles the sets and maps of the SETL programming language [Dewar, Schonberg and Schwartz 81]. A set of some elementary type is just an unordered collection of elements of the type. The usual set operators will be found in the next section. A table is much like a one or two dimensional array, but it may be sparse and have non-integer subscript types. The following gives a syntax for these

constructed types:

simple-type → *elementary-type* |
set of *elementary-type* |
table[*elementary-type*] of *simple-type* |
table[*elementary-type*][*elementary-type*] of *simple-type* |

Examples of these constructed types will be found in each subsequent section of this chapter.

Simple Primitives

These primitive functions take zero or more parameters, and return a value of a simple type. Some are generic in that some parameters need not be of any particular simple type. Such parameters will be shown as type **any-type**. Many of the primitives are familiar, like those needed to determine the current time and perform the usual arithmetic, set, and boolean operations.

A few of the primitives perform the cryptographic functions which were introduced in Chapter II and formalized in Chapter III. Functions are defined which create seeds, create keys and partial keys from seeds, and merge partial keys. The following identity provides an example of the use of the partial key primitives. It simply asserts that partial-keys formed from a key using a common seed can be merged back into the original key.

if s = *create-seed*() then
m=*merge-partials*(*form-partial*(1, s, m, 2), *form-partial*(2, s, m, 2))

The following provides detailed definitions of the primitive functions.

create-seed()→**seed**
Returns a seed derived from a physically random process within the instant node, and has no parameters.

create-public (*s*:**seed**)→**public-key**
Returns a public key that is a function of the parameter, seed *s*.

create-private (*s*:**seed**)→**private-key**
Returns a private key that is a function of the seed *s*. The private key corresponds to the public key created by a call to *create-public* with the same parameter *s*.

form-partial (*n*:**any-type**, *s*:**seed**, *a*:**any-type**, *m*:**integer**)→**partial-key**
Returns a partial value of the parameter *a*, with a threshold value of *m* (see *merge-partials*), using seed *s*. Calls with different values or types for *n* produce distinct partial values. *m* different partial values created with identical *s* are necessary and sufficient to determine the original value *a*. The seed *s* can not be determined even if all results of all possible calls are available, and without the seed the values of any call give no clue about the values of *a* used in another call.

merge-partials (*p*:set of **partial-key**)→*a*:**any-type**
Returns the original value of *a* which was divided into parts by *form-partial*. The parameter *p* must include at least as many partials formed from the original *a* as the threshold with which they were formed.

compress(*a*:**any-type**)→*i*:**integer**
Returns a cryptographic compression of the argument into an integer. Thus, given *a* and $i = compress(a)$ and the function *compress*, it is infeasible, under the assumptions of Chapter III, for an adversary to produce a' such that $i = compress(a')$ and $a' \neq a$.

now()→**time** Returns the time maintained by the clock of the instant node.

suicide (*m*:**integer**)
A real-time counter is set to count down for an interval of *m*, and if the counter ever reaches *0*, the instant vault sets all its secret *V*-functions to the value *erased* and in effect kills itself.

cardinality (*s*:set of **any-type**)→**integer**
Returns the number of distinct members of the set *s*.

+, −, ×→**integer** These are the usual infix operations performed on integers. Also − applied to two **times** is an **integer** which is negative when the time on the right is before the time on the left. (See definition of **time**.)

$-$, $\cup$, $\cap$ →set of **any-type**
The usual infix operators defined on sets, returning sets.

$<$, $\leq$, $=$, $\geq$, $>$, $\neq$ →**boolean**
Comparison infix operators.

$\in$, $\notin$, $\subseteq$ →**boolean**
Set membership, its negation, and subset.

Simple Constants

Besides the standard use of Arabic numerals as literal constants, there are two major sorts of constants used in the specification language. One kind of constant is used to indicate the various vacuous values, such as the empty set, uninitialized or don't-care values, and a special value indicating that all information about any previous value of the function is lost. The second sort of constant is used to reference information certified into the vault initially which specifies the keys, number and quorum sizes of the two groups of trustees and the enforced delay intervals on their actions. The certification of constant values into vaults is covered in Chapter IX.

Of course more elaborate versions of the algorithms presented here might include mechanisms to allow some or all of the constant values related to the trustees to be changed during operation of the network—much as the *SET_MINIMA* O-function does in the present algorithms. But such flexibility may actually prove undesirable, since those supplying information to a system may not wish to do so if the ground rules for its security can be revised in an arbitrary way.

A detailed definition of the simple constants follows:

empty The empty set.

undefined No particular value.

erased No trace or clue is left about the previous value of any *V*-function with this value.

cooling-off-interval
The minimum interval of time required between the time the last member of a majority of present nodes signs a proposal and the time the first node signs the announcement of the action defined by that proposal.

trustee-1-publics
The set of public keys held by the trustees at level 1 which are used to check all signatures purported to be made by trustees at level 1.

trustee-2-publics
The set of public keys held by the trustees at level 2.

trustee-1-quorum
The number of trustees at level 1 whose signatures are sufficient to authorize anything that can be authorized by trustees at level 1.

trustee-2-quorum
The number of signatures of trustees at level 2 required to authorize any proposed action. Also the number of trustees at level 2 whose trustee partials are required by the replacing node in a restart.

trustee-1-ids The set of **node-ids** which includes one member for each trustee at level 1. (As mentioned elsewhere, trustees are not nodes, but this convention greatly reduces the proliferation of types and typing mechanisms.)

trustee-2-ids The set of **node-ids** which includes one element for each trustee at level 2.

§3 Secret *V*-functions

The *V*-functions which record information not publicly available are defined, their use discussed, and initial values given.

*V*ariable functions, or *V*-functions, are the variables which hold a vault's state. The *V*-functions of a vault can be divided into those which the vault must keep secret and those which are public knowledge. This section presents the secret *V*-functions; the next section presents the non-secret *V*-functions.

The *V*-function definitions presented here usually include three parts: (1) a heading which defines the name and type of the *V*-function; (2) an initial value

part that includes the name and an expression whose value is the initial value; and (2) a comment part which discusses the intended use of the *V*-function.

The following productions give the basic idea of the syntax, further details being supplied in later sections:

> ***v-function*** → ***name*** :***simple-type*** :V-function ***initial-value*** ***comment***
> ***initial-value*** → Initial: ***name*** = ***expression*** | ***derivation***
> ***comment*** → Comment: ***wildcard***

Vaults must at minimum maintain the secrecy of their private keys upon which the security of the entire system relies. There will be two different kinds of secret keys, as mentioned in the previous chapter. Some keys need never be known outside the vault—these are the node secret keys. Other keys are kept secret by the vault, but they have been divided into partial keys and provided to other vaults for use during a restart—these are the application secret keys. In the following two subsections, each kind of secret *V*-functions is considered separately.

Node Secret *V*-functions

The *V*-functions described in this subsection never leave the vault. When the vault destroys its own information content, the values of these *V*-functions are set to ***erased***.

This sub-section makes the first formal reference to the notion of sub-partial keys. These are just partials of partial keys. In other words, some threshold of sub-partial keys are sufficient to reconstruct the original partial key from which the sub-partials were originally formed. The algorithms in this chapter allow the trustees to decide how many, if any, sub-partial keys will be

used by the network. The reason for this is that while the use of sub-partials does provide somewhat more convenience and flexibility in the operation of the network, they also have non-trivial cost in terms of system resources (see Chapter VIII for analysis of resource requirements). Sub-partial keys allow a "quorum" of nodes to, among other things, cause any node not participating in the last key change to become "participated" and enter a state equivalent to that which would have been achieved had it participated in the key change, restart nodes in an arbitrary order, and diminish the quorum size. The essence of this mechanism is that sufficient sub-partial keys allow every quorum of "present" nodes to form a partial key for other nodes in the network.

The following are definitions of the node secret *V*-functions:

NODE_PRIVATE:**private-key**: *V*-function

Initial value: *NODE_PRIVATE* =
create-private (let *INITIAL_NODE_SEED* = *create-seed*())

Comment: The private application key of the instant node. The initial value uses a *V*-function which is local to the initialization process *INITIAL_NODE_SEED*.

NEW_NODE_PRIVATE:**private-key**: *V*-function

Initial value: *NEW_NODE_PRIVATE* = *undefined*

Comment: Returns the application private key which will be assumed by the instant node if it is a participant in a *CHANGE_KEYS* or subject of a *PARTICIPATE* before the next key change. This private key is created by *CREATE_KEYS* and corresponds with *NEW_NODE_PUBLIC*.

PARTIAL_SEED:**seed**: *V*-function

Initial value: *PARTIAL_SEED* = *undefined*

Comment: Returns the randomly created seed used to form partial keys. Created and changed by *CREATE_KEYS*, *PARTIAL_SEED* is used by *ISSUE_NEW_PARTIALS* and also by the subject node of *PARTICIPATE*.

PARTIAL_KEYS:table[**node-id**] of **partial-key**: *V*-function

Initial value: $\forall p$ *PARTIAL_KEYS*[p] = *undefined*

Comment: The partial key held by the instant node for the participated node p is *PARTIAL_KEYS*[p]. The constituent partial keys are received by *RECEIVE_NEW_PARTIALS* and by *RECEIVE_NEW_PARTICIPANT*.

NEW_PARTIAL_KEYS:table[**node-id**] of **partial-key**: *V*-function

Initial value: $\forall n$ *NEW_PARTIAL_KEYS*[n] = *undefined*

Comment: Returns the new partial key held by the instant node for the selected node. The value is obtained by *RECEIVE_NEW_PARTIALS* and will replace *PARTIAL_KEYS* iff the instant node participates in a *CHANGE_KEYS* before the next *CREATE_KEYS*.

SUB−PARTIALS:table[**node-id**][**integer**] of **partial-key**: *V*-function

Initial value: $\forall p \; \forall i$ *SUB−PARTIALS*[p][i] = *undefined*

Comment: The partial partial key held by the instant node for the participated node n, to be released to the node assuming the ith set of sub-partials. The values are obtained from *NEW_SUB−PARTIALS* after the instant node participates in a *CHANGE_KEYS*, or from the input supplied to *NEW_PARTICIPANT_RECEIVE*. The *SUB−PARTIALS*[p][i]s held by a quorum of present nodes for a particular set of sub-partials indexed by i are sufficient to allow *merge-partials* to determine a partial for node p.

NEW_SUB−PARTIALS:table[**node-id**][**integer**] of **partial-key**: *V*-function

Initial value: $\forall n \; \forall i$ *NEW_SUB−PARTIALS*[n][i] = *undefined*

Comment: Returns values accumulated since the last *CREATE_KEYS* which will replace *SUB−PARTIALS* iff the instant node participates in a *CHANGE_KEYS* before another *CREATE_KEYS*.

OWN_TRUSTEE_PARTIALS:table[**node-id**] of **partial-key**: *V*-function

Initial value: $\forall n$ *OWN_TRUSTEE_PARTIALS*[n] = *undefined*

Comment: *OWN_TRUSTEE_PARTIALS*[n] is a private key which must be present in the instant node when the instant node is the replacing node in a *RESTART* in which node n is the replaced node. Values of *OWN_TRUSTEE_PARTIALS* are obtained by the subject of *CERTIFY* for all the nodes it is certified for for (except itself), and any values for which the subject is not certified are erased. In an application where some different nodes have access to different data, a particular vault may not be approved to restart some nodes.

Application Secret *V*-functions

Care is taken to ensure that *APPLICATION_PRIVATE* can be recovered only with partial keys of the most-recently completed key change, and that *NEW_APPLICATION_PRIVATE* can be recovered with partial keys distributed for the next key change. Of course there is presumably much secret application data which must be included in checkpoints, and it should also be divided into current change period and new period—so that obsolete application data becomes inaccessible once a node changes keys. The aggregate *V*-function, *APPLICATION_SECRET_V-FUNCTIONS*, is assumed to contain all application secret data from the current change period; the aggregate *NEW_APPLICATION_SECRET_V-FUNCTIONS* contains all application data for the forthcoming key period.

The following are definitions of the two application *V*-functions relevant here, one for each aggregate:

APPLICATION_PRIVATE:**private-key**:*V*-function
Initial value: *APPLICATION_PRIVATE* = *create-private*(*create-seed*())
Comment: The private application key of the instant node.

NEW_APPLICATION_PRIVATE:**private-key**:*V*-function
Initial value: *NEW_APPLICATION_PRIVATE* = *undefined*
Comment: Returns the application private key which will be assumed by the instant node if it is a participant in a *RESTART* or subject of a *PARTICIPATE* before the next *CHANGE_KEYS*. This private key is created by *CREATE_KEYS* and corresponds with *NEW_NODE_PUBLIC*.

§4 Non-Secret *V*-functions

Those *V*-functions are presented which relate to node state that is not secret.

Some *V*-functions in this section are defined in terms of expressions involving other *V*-functions, and they have a "derivation" part instead of an initial value part:

derivation → Derivation: ***name = expression***

The *OWN_NODE* *V*-function is special in that its value never changes during the life of a node, but the actual initial value of each node's *OWN_NODE* must be unique. No initial value part or derivation is used for *OWN_NODE*.

As will be seen in Chapter VII, it is quite useful to distinguish those *V*-functions whose values must be in agreement across nodes, from those *V*-functions which are not subject to any consensus constraint. These two kinds of *V*-functions are covered in separate subsections.

Consensus *V*-functions

The non-secret *V*-functions presented in this subsection are intended to have identical value for all nodes with the same value of *CYCLE* (which is defined in the next subsection). They define the status of the network. As a notational convenience, the consensus *V*-functions are denoted collectively as *CONSENSUS_V-FUNCTIONS*.

NODES_IN_USE:set of **node-id**: *V*-function

Initial value: *NODES_IN_USE* = *empty*

Comment: Returns the set of node ids which includes an id for every node in the network. These exclude all removed nodes and include the newly *CERTIFY*ed initiate nodes which have not ever been members of *PARTICIPATED*, and all veteran nodes which are those nodes who have been members of *PARTICIPATED* at least once, whether or not they are presently participated.

USED_NODE_IDS:set of **node-id**: *V*-function

Initial value: *USED_NODE_IDS* = *trustee-1-ids* ∪ *trustee-2-ids*

Comment: Returns a set of node ids which are not suitable for use by any new node. *CERTIFY* ensures that new nodes do not use ids in *USED_NODE_IDS*, *REMOVE_NODES* places the id of all removed nodes into *USED_NODE_IDS*, and *RESTART* places the id of the replaced node in *USED_NODE_IDS*. For simplicity in typing and signature checking primitives, as already mentioned, **node-id**s are also used to identify the trustees.

PARTICIPATED:set of **node-id**: *V*-function

Initial value: *PARTICIPATED* = *empty*

Comment: Returns the set of node ids which includes exactly those nodes which were included as *PARTICIPANTS* in the last *CHANGE_KEYS* and all those nodes which have been the subject of subsequent *PARTICIPATE*s. Any node which is to become present must be a member of *PARTICIPATED*.

PRESENT:set of **node-id**: *V*-function

Initial value: *PRESENT* = *empty*

Comment: Returns the set of node ids which defines the most privileged and capable subset of nodes. Every *QUORUM* of members of *PRESENT* have sufficient partial keys to enable them to restart any present node. Signatures of a *QUORUM* of members of *PRESENT* are required before any node may perform any synchronized *O*-function.

ABSENT:set of **node-id**: *V*-function

Derivation: *ABSENT* = *NODES_IN_USE* − *PRESENT*

APPLIED:set of **node-id**: *V*-function

Initial value: *APPLIED* = *empty*

Comment: Returns the set of node ids of all nodes which currently have an application. Nodes enter applied when they are the subject of an *APPLY* or when they are the replacing node in a *RESTART* and they leave *APPLIED* when they are a subject of *REMOVE_NODES* or the replaced node of a *RESTART*.

MAJORITY:**integer**: *V*-function

Initial value: *MAJORITY = 0*

Comment: The minimum number of signatories required before any announcement or action can be carried out. Set by *CHANGE_PRESENT*. Must be at least as great as *QUORUM* and no greater than *cardinality*(*PARTICIPATED*), and satisfy the *MINIMUM_MARGIN* requirement.

MARGIN:**integer**: *V*-function

Derivation: *MARGIN* = (*2* × *MAJORITY*) − *cardinality*(*PRESENT*)

Comment: The minimum intersection between any two *MAJORITY*s of *PRESENT* nodes.

MINIMUM_MARGIN:**integer**: *V*-function

Initial value: *MINIMUM_MARGIN = 1*

Comment: The smallest allowable value of *MARGIN*. The value of *MINIMUM_MARGIN* is changed only by *SET_MINIMA*, and can not be set below 1, which ensures that *MAJORITY* is always a simple majority of *cardinality*(*PRESENT*).

MINIMUM_QUORUM:**integer**: *V*-function

Initial value: *MINIMUM_QUORUM = 0*

Comment: The smallest allowable value of *QUORUM*. Set by *SET_MINIMA*.

QUORUM:**integer**: *V*-function

Derivation: *QUORUM* = *QUORUMS*[*LAST_CHANGE*]

Comment: The current quorum.

QUORUMS:table[**integer**] of **integer**: *V*-function

Initial value: *QUORUMS*[*0*] = *0* ∧
∀*n*{ if *n* ≠ *1* then *QUORUMS*[*n*] = *undefined*}

Comment: Returns the number of partial keys required for a restart of a node who last participated during key change *n*, for all *n* ≤ *LAST_CHANGE*. Thus, *QUORUMS*[*LAST_CHANGE*] returns the number of partials of the current key change period which are required by *merge-partials*. And *QUORUMS*[*LAST_CHANGE* + *1*], returns the number of nodes whose partials or sub-partials will be required for a successful *merge-partials* during the next key change period, if no further *CREATE_KEYS* occurs before the next *CHANGE_KEYS*.

SUB−PARTIALS_REMAINING:table[**node-id**] of **integer**: *V*-function

Initial value: $\forall n$ *SUB−PARTIALS_REMAINING*[n] = *undefined*

Comment: Returns the number of sub-partials remaining for the nth node. New entries are established by *CERTIFY*.

NEW_SUB−PARTIALS_REMAINING:table[**node-id**] of **integer**: *V*-function

Initial value: $\forall n$ *NEW_SUB−PARTIALS_REMAINING*[n] = *undefined*

Comment: Returns the number of sub-partials that are needed by the nth node during the current key change period as established in the last *CREATE_KEYS*.

LAST_CHANGES:table[**node-id**] of **integer**: *V*-function

Initial value: $\forall n$ *LAST_CHANGES*[n] = *undefined*

Comment: *LAST_CHANGES*[n] returns the last key change period during which node n participated in the initial *CHANGE_KEYS* or in which n was the subject of a *PARTICIPATE*. New entries are established by *CERTIFY*.

LAST_CHANGE:**integer**: *V*-function

Derivation: *LAST_CHANGE* = *LAST_CHANGES*[i]
$\{\forall J \{$*LAST_CHANGES*[i] ≥ *LAST_CHANGES*[j]$\}\}$

Comment: Returns the number of the last key change the instant node has processed, whether or not the instant node participated.

KEY_CREATION_#:**integer**: *V*-function

Initial value: *KEY_CREATION_#* = *0*

Comment: Returns the serial number of action calls of the *CREATE_KEYS O*-function. Notice that there may be more than one call to *CREATE_KEYS* between calls to *CHANGE_KEYS* and that all but the last such call have no effect on the *CHANGE_KEYS* because *PARTIALS_RECEIVED* is emptied by *CREATE_KEYS* and all relevant transfers include the *KEY_CREATION_#*. (The multiple calls may be convenient since they allow a new quorum and complement of sub-partials to be selected.)

SUICIDE_INTERVAL:**integer**: *V*-function

Initial value: *SUICIDE_INTERVAL* = *cooling-off-interval*

Comment: Returns a time interval (i.e. an **integer**) during which a node must become participated or it will commit suicide. The actual call to *suicide* is made on the *SUICIDE_INTERVAL* minus the amount of time since the earliest timestamp among the majority of signatories to the *CHANGE_KEYS* or *PARTICIPATE* in which the instant node is a subject. The initial value is such that a *SET_MINIMA* must occur before a *cooling-off-interval* has elapsed since the first *CHANGE_KEYS* participated in.

NODE_PUBLICS:table[**node-id**] of **public-key**: *V*-function

Initial value: $\forall n$ *NODE_PUBLICS*[n] = *undefined*

Comment: The current application public key of every node in use.

APPLICATION_PUBLICS:table[**node-id**] of **public-key**: *V*-function

Initial value: $\forall n$ *APPLICATION_PUBLICS*[n] = *undefined*

Comment: The current node public key of every node in use. New entries are established by *CERTIFY*, and existing entries are changed for subjects of *CHANGE_KEYS* and *PARTICIPATE*.

CERTIFICATION:table[**node-id**] of set of **node-id**: *V*-function

Initial value: $\forall n$ *CERTIFICATION*[n] = *undefined* $\wedge$
CERTIFICATION[*OWN_NODE*] = *empty*

Comment: Each node in use n has associated with it a set of other nodes *CERTIFICATION*[n] whose applications it is allowed to assume, either by *APPLY* or *RESTART*. The nodes comprising the certification of a node are initialized and changed by *CERTIFY*.

PROPOSALS_PENDING:set of **integer**: *V*-function

Initial value: *PROPOSALS_PENDING* = *empty*

Comment: Returns the set of all cycle numbers of proposals which have been proposed but not canceled or carried out.

COMPRESSED_HISTORY:**integer**: *V*-function

Initial value: *COMPRESSED_HISTORY* = *0*

Comment: Returns a compression of *CONSENSUS_V-FUNCTIONS* formed before the action of the last cycle was completed. Since *COMPRESSED_HISTORY* is included in *CONSENSUS_V-FUNCTIONS*, *COMPRESSED_HISTORY* is a *V*-function of the entire series of states obtained by the identical *V*-functions during all previous cycles. Because *COMPRESSED_HISTORY* is checked in the input of every synchronized *O*-function, no node will perform any synchronized action unless its entire history of *CONSENSUS_V-FUNCTIONS* states is the same as every other node performing the action. This is largely a redundant mechanism; see Chapter VII.

Individual *V*-functions

Some of the non-secret *V*-functions presented in this subsection will have unique values, never obtainable by another node. For example, a node's record of its own past public keys will be unique. Other *V*-functions covered here may

have nearly the same values across nodes, but this strict consensus is not enforced as in the previous subsection. For example, *PARTIALS_RECEIVED_FROM* contains node ids of all the nodes from which a node has received partial keys. These may vary as the partial keys are received in different orders and possibly from different sets of nodes, but those maintained by all participated nodes will ultimately include node ids from all participated nodes. Just as *CONSENSUS_V-FUNCTIONS* was used to denote the entire collection of consensus *V*-functions, *INDIVIDUAL_V-FUNCTIONS* will be used to denote the collection of individual *V*-functions.

OWN_NODE:**node-id**: *V*-function

Comment: Returns the **node-id** which identifies the instant node for its entire life. The value should be distinct from that of all other nodes, so that *CERTIFY* will allow the node to be initiated into the network. Examples of possible actual implementation values include the simple serial numbers of a node or the initial node public key.

PHASE:*1..2*: *V*-function

Initial value: *PHASE = 1*

Comment: Returns the current phase, either *1* or *2*, which is used by all synchronized *O*-functions. When *PHASE = 1* a node will add its signature to any announcement or action which has insufficient signatures and does not raise an exception, then the node will change to *PHASE = 2*. When *PHASE = 2* a node will not add its signature to any announcement. In either phase, when a node receives an announcement with sufficient signatures and no exception is raised, it will perform the effects section, which includes setting *PHASE = 1* and incrementing *CYCLE*.

CYCLE:**integer**: *V*-function

Initial value: *CYCLE = 1*

Comment: The basis of all synchronization of the network, this monotonically increasing value ensures that all nodes will process synchronized *O*-functions in exactly the same order. Returns the serial number of the next announcement or action which the present node has yet to perform.

NEW_NODE_PUBLIC:**public-key**: *V*-function

Initial value: *NEW_NODE_PUBLIC*: = *undefined*

Comment: Returns the instant node's own new node public key, which corresponds with *NEW_NODE_PRIVATE*, and whose value was determined during the last *CREATE_KEYS*.

NEW_APPLICATION_PUBLIC:**public-key**: *V*-function

Initial value: *NEW_APPLICATION_PUBLIC*: = *undefined*

Comment: Returns the instant node's own new application public key, which corresponds with *NEW_APPLICATION_PRIVATE*, and whose value was determined during the last *CREATE_KEYS*.

ALL_OWN_NODE_PUBLICS:set of **public-key**: *V*-function

Initial value: *create-public* (*INITIAL_NODE_SEED*) ∈ *ALL_OWN_NODE_PUBLICS*

Comment: Returns all the node public keys that have been used by the instant node to sign proposals which are pending. Because the number of proposals pending can be kept from growing too large, through the use of *CANCEL_PROPOSAL*, *cardinality*(*ALL_OWN_NODE_PUBLICS*) can be kept to a modest size. *INITIAL_NODE_SEED* is a variable which is local to the initialization and which is defined in the description of *NODE_PRIVATE*.

PARTIALS_RECEIVED_FROM:set of **node-id**: *V*-function

Initial value: *PARTIALS_RECEIVED_FROM* = *empty*

Comment: Returns the set of nodes for which the instant node has received partial keys during the current key creation period. This *V*-function is emptied by *CREATE_KEYS*, and new members are added to it by *RECEIVE_NEW_PARTIALS*, *NEW_PARTICIPANT_RECEIVE* and *RECEIVE_NEW_PARTICIPANT*. The unsynchronized *O*-function *PARTIALS_RECEIVED* issues signed statements of minimum content of *PARTIALS_RECEIVED_FROM*. These statements must be received from all nodes who participate in a *CHANGE_KEYS*, and they must include every such participating node. The statements are also checked for by *CHANGE_PRESENT* to ensure that all nodes made present have partial keys from all other nodes made present, which ensures that all necessary *RECEIVE_NEW_PARTICIPANT* and *NEW_PARTICIPANT_RECEIVE*s have completed for any *PARTICIPATE*ed nodes.

§5 Templates, Template Types, & Primitives

Input and output parameter passing mechanisms are described which include constructed descriptions of hierarchically encrypted data, and primitives for performing cryptographic operations on data.

An unusually powerful parameter mechanism has been incorporated into the specification language used here, for several reasons. First, it allows the underlying structure of multiply encrypted messages to be shown clearly. Second, it allows much of the routine checking and cryptographic transformations to be handled cleanly, and without complicating the rest of the algorithm description unnecessarily. Third, the particular form used here can also provide descriptive names, types, and sometimes values for the parts of parameters.

Templates

The basic syntax for the parameter description mechanism, called a *template*, is shown in the following productions:

template → *name*:*construction*
construction → **constructor-type* <*item-list*> |
constructor-type <*item-list*>
constructor-type → signed | sealed | signatured
item → *expression* = *name*:*type* | *name*:*type* |
expression = :*type* | *name*: | :*type*
item-list → *item-list* , *item* | *item*
type → *simple-type* | *construction*

The constructor types are covered in the next subsection. A * denotes a part of a template, or an entire template, that is optional. The rules for when the optional parts must appear in input, and when they are output are covered in the subsection on template primitives. the names which may appear in a template serve as the formal parameters. An item in a template may include an expression. When an expression provides a value for an item in a template describing input, the actual parameter supplied must have the identical value; when an expression provides a value for an item in a template describing output, the value provided is output.

Notice that all five non-empty possible combinations of the three components of an ***item*** can be used in a template. One form of ***item*** is ***name*** :***type***. It is the usual formal parameter when used for input, and is used to return the value of the formal parameter (which must be of the specified type) in an output template. Another form of ***item*** is ***expression*** = :***type***, which is used in an input template to cause an initial "bad template" exception if the corresponding input actual parameter does not have the value of the expression. It is used in an output template to return a value for which a parameter name is not needed. The most elaborate form is ***expression*** = ***name*** :***type***. It serves the same function as the previous form, except that a parameter name is associated with the value. When only a name is supplied, ***name*** :, the type and value are obtained from another item with the same name. When only a type is supplied, :***type***, the value of the parameter is ignored.

Several items or even whole templates in an *O*-function may share the same name. Items with the same name must have the same value. Templates with the same name are just different copies of the same template. The next section contains a number of templates which may serve as instructive examples.

Template Types

The three template types were shown in the formalism of the previous subsection as ***constructor-type***. This subsection gives a detailed description of each, but these descriptions are best taken together with those of the template primitives of the following subsection.

signed A digital signature of a structure of constituent elements. (See the primitives *sign*, and *check-signature*.)

signatured A collection of digital signatures of the same material. There are several possible implementations of the notion of **signatured**, such as repeated encryption of a single bit string, individually signed seperate copies of the same bit string, signatures made on a compression of the matter to be signed, or a combination of these approaches. It may also be desirable to explicitly include in the **signatured** some bits indicating who has made each signature. (See the primitives *sign* and *check-signatured*.) A **signatured** may also include timestamps provided by the signatories. (See the primitives *latest-signature* and *earliest-signature*.)

sealed An encrypted form of the constituent elements of a structure. These should include a random component, as described in Chapter II. (See the primitives *seal* and *unseal*)

Template Primitives

The following primitive functions are used to perform cryptographic transformations on input and output of *O*-functions. Input parameters which are included in a **signatured**, **signed** or **sealed** construction must be the subject of a *check-signatured*, *check-signature* or *unseal* primitive respectively if the constituent items of the construction are to be accessed. Once the primitive is applied, free use can be made of the items of the construction. The omission of optional input constructions in an "actual parameter" (those marked by a * in the "formal parameter") which are the subject of *check-signatured* or *check-signature* cause these primitives to return *false*. Optional output constructions are output if and only if their **signed** construction is the subject of a *sign* primitive. Any **signatured** constructions appearing as input will be output with an additional signature if they are the subject of a *sign* primative—even though the construction name does not appear in an output section.

The following identity provides an example of some of the template primitive functions. It simply asserts that sealing and signing are inverses when keys created from the same seed are used.

if s = *create-seed*() then

m=*unseal*(*seal*(m, *create-public*(s)), *create-private*(s))

The following are definitions of the template primitives:

sign(**signed**<a:**any-type**, · · · >, k:**private-key**)→
Optional output parameters are output iff their **signed** structure is the subject of a *sign* primitive.

seal(**sealed**<a:**any-type**, · · · >, k:**public-key**)→
Must be applied to any output structure that is of type **sealed**, if that structure will be included in *O*-function output. The public key k is used to perform the encryption.

unseal(**sealed**<a:**any-type**, · · · >, k:**private-key**)→
Makes accessible (but does not actually return) the unsealed, i.e. un-encrypted, form of the input structure s iff s was the output of an *O*-function which resulted from a *seal* primitive applied with the public key corresponding to the private key k.

check-signature (s:**signed**<**any-type** · · · >, k:**public-key**)→**boolean**
Checks the digital signature of the subject input structure s by decrypting it with the public key k and checking for the redundancy required by convention, and returns *true* iff the signature passes the test.

check-signatured(s:**signatured**<**any-type** · · · >, k:set of **public-key**, m:**integer**)→
boolean
Returns *true* iff a set of digital signatures of the subject input structure s can be checked as having been formed by holders of m private keys corresponding to m of the public keys contained in the set of keys k (i.e. $\exists p$:set of **public-key** {*cardinality*(p) = m ∧ $p \subseteq k$ ∧ $\forall n$:**public-key** { if $n \in p$ then *check-signature*(s, n)}}

latest-signature (s:**signatured**<**any-type** · · · >)→**time**
Returns the most recent timestamp contained in the signatures of s.

earliest-signature (s:**signatured**<**any-type** · · · >)→**time**
Same as *latest-signature* except the time of the earliest.

§6 Synchronized *O*-functions

Presents the remainder of the specification language and uses it to define the major *O*-functions of the proposed design.

The *O*-functions presented in this section define all the synchronized actions performed by the network. These allow for consensus by the nodes on the state of the network, and implement all the changes in network status. Figure 1 shows the *O*-functions which change the status of individual nodes, such as by certifying them into the network, removing nodes from the network, and restarting a disabled node. The *CHANGE_KEYS* *O*-function of the figure allows a set of nodes to each change their public keys and receive new partial keys from the other nodes, once the new partials have been sealed with the receiving node's new keys. One other *O*-function, not shown in the figure, has an impact on the network status. It establishes the minimum values of important system parameters.

Properties of the synchronization mechanism are demonstrated in Chapter VII. For the present purposes, it is important to notice that synchronization is provided by a cycle counter, *CYCLE*, maintained by each node. Each node can perform the action of only one synchronized *O*-function call for each successive cycle. A majority of present nodes must each sign a template which defines every synchronized *O*-function call and the numbered cycle during which it is to be performed. No node signs more than one template during a single cycle. This arrangement ensures that nodes perform exactly the same *O*-function call during each cycle number. In particular, the *CONSENSUS_V-FUNCTIONS* of all nodes in a particular cycle are guarantee to be identical.

Chapter II gave a description of three levels of trustees: trustees at level 1 are not in a position to compromise system security, but are able to perform the day to day operations necessary to ensure the system's reliability; trustees

at level 2 establish policy and make security relevant decisions; trustees at level 3 are not part of the mechanism of this chapter, but are considered in Chapter VII, as mentioned above. The present section is divided into those *O*-functions callable by trustees at level 1, and those callable by trustees at level 2. Before any trustee level 2 *O*-function call can be made, however, it must be proposed: the definition of the security relevant parts of the call must be included in a trustee level 1 call to *PROPOSE*, which takes up one cycle. After this call has been made, a delay of length *cooling-off-interval* is enforced before the trustee level 2 action can be performed, by the corresponding trustee level 2 call. Any other actions may occur during intermediate cycles, and the trustee 2 level call can be blocked from ever occurring by the *CANCEL_PROPOSAL* synchronized *O*-function. The following two subsections provide the details of each of these two kinds of synchronized *O*-functions. Before the *O*-functions can be presented, however, the remainder of the specification language must be described.

O-function Syntax and Semantics. *O*-functions are composed of five major parts, roughly following the the structure put forward by Parnas [72]. For the purposes of the present work, the simple input parameter list of the Parnas notation has been extended into optional input and output parts, which use the template mechanism described in the previous section. The third part of an *O*-function is merely for documentation. The fourth part lists a series of named exception conditions, all of which are checked sequentially. If all the checks are successful, then the effects part (the fifth part) is performed.

Some of the statements which make up the effects part are boolean expressions. They have the effect of changing their constituent *V*-functions or formal parameters to values which make the expression true. Other statements do not return values, but rather are composed of primitive functions with side effects. There is no implied sequential order of execution. All values of *V*-functions within the effects part represent the value of the *V*-function after the entire *O*-

function is completed. Those *V*-functions whose names are enclosed in single quotes represent the value of the *V*-function before the call to the *O*-function. The following productions give the syntax of *O*-function definitions and their five parts:

O-function → *header input output comment exceptions effects* |
header input comment exceptions effects |
header output comment exceptions effects

header → *name* :*O*-function

input → Input: *template*

output → Output: *template*

comment → Comment: *wildcard*

exceptions → Exceptions: *exception-list*

exception-list → *exception-list* ,*exception* | *exception*

exception → *name* : *boolean-expression*

effects → Effects: *statement*

statement → *boolean-expression* | {*statement-list* } |
if *boolean-expression* then *statement* |
if *boolean-expression* then *statement* else *statement*
with *name* [*expression*]*statement*

statement-list → *statement-list* , *statement* | *statement*

boolean-expression → ¬*boolean-expression* |
(*boolean-expression*) |
boolean-primitive-function (*expression-list*) |
expression predicate expression |
if *boolean-expression* then *boolean-expression* |
if *boolean-expression* then *boolean-expression*
else *boolean-expression*
quantifier name :*elementary-type* {*boolean-expression* } |
quantifier name :*elementary-type*
quantifier name :*elementary-type* {*boolean-expression* }

expression → *name* | '*name* ' | *expression operator expression* |
(*expression*) | *name* [*expression*] |
name [*expression*][*expression*] |
let *name* = *expression* |
with *name* [*expression*]*expression* |
primitive-function (*expression-list*)

expression-list → *expression-list* , *expression* | *expression*

The keyword "let" is used to establish temporary variables within *O*-functions to avoid re-writing long expressions. The keyword "with" is used,

much as in some programming languages, to extend the qualification of a name (in this case, a part of a construction selected by a particular index) over an expression.

Trustee 2 *O*-functions

There are three trustee level 2 *O*-functions. The *CERTIFY* function is used to bring new nodes into the network, as can be seen in Figure 1. This function is critical to the security of the entire system because if sufficient corrupted or even subverted nodes (see Chapter III) are brought into the network, then many of the security measures are useless. It can also be used to establish and modify a set, for each non-applied node, of nodes that the node can replace during a restart. (This might be used in an application where some nodes have data so sensitive that some vaults should never be able to access it.)

The *SET_MINIMA* function is also very important. It establishes the minimum margin (the significance of which is discussed in Chapter VII), the minimum quorum of present nodes required for system operation, and the amount of time a node will wait to participate before it erases its own secret *V*-function values. All three of these parameters determine the difficulty of the various attacks which could be perpetrated against the system.

The final level 2 function is *REMOVE_NODES*. It simply allows nodes to be taken out of the network, as illustrated in Figure 1.

One thing to notice about these function definitions is that some of the latter exceptions and initial effects are the same. These common mechanisms are used to establish synchronization. When one of these functions is called and the *ANNOUNCEMENT* template does not have signatures from a majority of present nodes (and the present node has not added its signature to an announcement of the current cycle), then the node simply adds its signature to the announce-

ment and returns; when one such function is called and there are sufficient signatures on the announcement, the node changes to phase 1 of the next cycle and performs the required action. Thus, to perform a particular synchronized action as a particular cycle, at least a majority of present nodes in phase 1 of that cycle must first be called to obtain sufficient signatures on the desired announcement, and then this announcement can be used in subsequent calls to cause any node to perform the synchronized action.

The following are the detailed function definitions:

CERTIFY: *O*-function
Input:
ANNOUNCEMENT:**signatured**
<*NODE_CERTIFIED*:**node-id**,
NODE_KEY:**public-key**,
APPLICATION_KEY:**public-key**,
NODES_RESTARTABLE:set of **node-id**,
TRUSTEES_SUPPLYING:set of **node-id**,
TRUSTEES'_PARTIALS:table[**node-id**] of
TRUSTEE_PARTIALS:**sealed**
<:table[**node-id**] of **partial-key**>,
PROPOSAL_CYCLE_#:**integer**,
CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
certify = :**announcement-kind**>,
PROPOSAL:**signatured**
<*NODE_CERTIFIED*:**node-id**,
NODE_KEY:**public-key**,
APPLICATION_KEY:**public-key**,
NODES_RESTARTABLE:set of **node-id**,
LATEST_TIMESTAMP:**time**,
PROPOSAL_CYCLE_#:**integer**,
propose-certify = :**proposal-kind**>
Comment: The set of nodes the *NODE_CERTIFIED* is allowed to restart is changed to *NODES_RESTARTABLE*. If the *NODE_CERTIFIED* node id is not in *NODES_IN_USE*, then it becomes included in *NODES_IN_USE*, and the *NODE_KEY* and *APPLICATION_KEY* parameters input are used to establish table entries corresponding to the new node. All nodes change the set of nodes the *NODE_CERTIFIED* is allowed to restart to *NODES_RESTARTABLE*. If a node's own id appears in its *NODES_RESTARTABLE* then it is allowed to be

*APPLY*ed. The *NODE_CERTIFIED* recovers the *OWN_TRUSTEE_PARTIALS* that are needed by merging the *TRUSTEE_PARTIALS* input by a *trustee-2-quorum*. The *OWN_TRUSTEE_PARTIALS* that are no longer needed are erased.

Exceptions:

BAD_NODE_CERTIFIED: *NODE_CERTIFIED* ∈ *USED_NODE_IDS*

PROPOSAL_NOT_PENDING: *PROPOSAL_CYCLE_#* ∉ *PROPOSALS_PENDING*

INSUFFICIENT_TRUSTEE_2_SIGNATURES: ¬*check-signatured*
(*ANNOUNCEMENT_DEFINITION*, *trustee-2-publics*, *trustee-2-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = 2 ∧
¬(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT*, {∀*n*:**node-id**
{ if *n* ∈ *PRESENT* then *NODE_PUBLICS*[*n*]}}, *MAJORITY*))

INSTANT_NOT_SIGNATORY: ¬∃ *K*:**public-key** {*k* ∈ *ALL_OWN_NODE_PUBLICS* ∧
check-signature (*PROPOSAL*, *k*)}

TOO_EARLY: *now* − *LATEST_TIMESTAMP* < *cooling-off-interval*

Effects:

if ¬*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign* (*ANNOUNCEMENT*, *NODE_PRIVATE*), *PHASE* = 2}
else {
PROPOSAL_CYCLE_# ∉ *PROPOSALS_PENDING*,
CYCLE = '*CYCLE*'+1,
PHASE = 1,
COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),
CERTIFICATION[*NODE_CERTIFIED*] = *NODES_RESTARTABLE*,
if *NODE_CERTIFIED* ∉ '*NODES_IN_USE*' then
{*NODE_CERTIFIED* ∈ *NODES_IN_USE*,
APPLICATION_PUBLICS[*NODE_CERTIFIED*] = *APPLICATION_KEY*,
NODE_PUBLICS[*NODE_CERTIFIED*] = *NODE_KEY*,
LAST_CHANGES[*NODE_CERTIFIED*] = 0,
SUB−PARTIALS_REMAINING[*NODE_CERTIFIED*] = 0},
if *NODE_CERTIFIED* = *OWN_NODE* then
{∀*k*:**node-id**{ if *k* ∈ *TRUSTEES_SUPPLYING* then
unseal(*TRUSTEES'_PARTIALS*[*k*], *NODE_PRIVATE*)},
∀*r*:**node-id** { if *r* ∈ *NODES_RESTARTABLE* ∧
r ∉ '*CERTIFICATION*'[*OWN_NODE*] then
OWN_TRUSTEE_PARTIALS[*r*] = *merge-partials*(
{∀*k*:**node-id**{ if *k* ∈ *TRUSTEES_SUPPLYING* then
with *TRUSTEES'_PARTIALS*[*k*]
{*TRUSTEE_PARTIALS*[*r*]}}}),
∀*n*:**node-id** { if *n* ∈ '*CERTIFICATION*'[*OWN_NODE*] ∧
n ∉ *NODES_RESTARTABLE* then
OWN_TRUSTEE_PARTIALS[*n*] = *erased*}}}

SET_MINIMA: *O*-function

Input:

ANNOUNCEMENT:**signatured**

<*NEW_MINIMUM_QUORUM*:**integer**,

NEW_MINIMUM_MARGIN:**integer**,

NEW_SUICIDE_INTERVAL:**integer**,

PROPOSAL_CYCLE_#:**integer**,

CYCLE = *CYCLE_#*:**integer**,

COMPRESSED_HISTORY = :**integer**,

set-minima = :**announcement-kind**>,

PROPOSAL:**signatured**

<*NEW_MINIMUM_QUORUM*:**integer**,

NEW_MINIMUM_MARGIN:**integer**,

NEW_SUICIDE_INTERVAL:**integer**,

LATEST_TIMESTAMP:**time**,

PROPOSAL_CYCLE_#:**integer**,

propose-set-minima = :**proposal-kind**>

Comment: The *V*-functions holding the minimum values are changed to the values of the corresponding parameters. The new minima must not be larger than a possible current actual as opposed to minimum value.

Exceptions:

NEW_MINIMUM_MARGIN_TOO_SMALL: *NEW_MINIMUM_MARGIN* < *1*

NEW_MINIMUM_MARGIN_TOO_BIG: *NEW_MINIMUM_MARGIN* > *MARGIN*

NEW_MINIMUM_QUORUM_TOO_BIG: *NEW_MINIMUM_QUORUM* > *QUORUM*

PROPOSAL_NOT_PENDING: *PROPOSAL_CYCLE_#* ∉ *PROPOSALS_PENDING*

INSUFFICIENT_TRUSTEE_2_SIGNATURES: ¬*check-signatured*
(*ANNOUNCEMENT_DEFINITION*, *trustee-2-publics*, *trustee-2-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = *2* ∧
¬(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT*, {∀*n*:**node-id**
{ if *n* ∈ *PRESENT* then *NODE_PUBLICS*[*n*]}}, *MAJORITY*))

INSTANT_NOT_SIGNATORY: ¬∃ *K*:**public-key** {*k* ∈ *ALL_OWN_NODE_PUBLICS* ∧
check-signature (*PROPOSAL*, *k*)}

TOO_EARLY: *now* − *LATEST_TIMESTAMP* < *cooling-off-interval*

Effects:

if ¬*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT*, *NODE_PRIVATE*), *PHASE* = *2*}

else {

PROPOSAL_CYCLE_# ∉ *PROPOSALS_PENDING*,

CYCLE = '*CYCLE*'+*1*,

PHASE = *1*,

COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),

MINIMUM_QUORUM = *NEW_MINIMUM_QUORUM*,
MINIMUM_MARGIN = *NEW_MINIMUM_MARGIN*,
SUICIDE_INTERVAL = *NEW_SUICIDE_INTERVAL*}

REMOVE_NODES: *O*-function
Input:
ANNOUNCEMENT:**signatured**
<*NODES_TO_REMOVE*:set of **node-id**,
PROPOSAL_CYCLE_#:**integer**,
CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
remove = :**announcement-kind**>,
PROPOSAL:**signatured**
<*NODES_TO_REMOVE*:set of **node-id**,
LATEST_TIMESTAMP:**time**,
PROPOSAL_CYCLE_#:**integer**,
propose-remove = :**proposal-kind**>
Comment: The node ids of the *NODES_TO_REMOVE* are removed from *NODES_IN_USE*, and all secret table entries for the *NODES_TO_REMOVE* are erased. The removed nodes commit suicide.
Exceptions:
NO_SUCH_NODE_IN_USE: ¬ *NODES_REMOVED* ⊆ *NODES_IN_USE*
REMOVING_PRESENT: ∃*n*:**node-id** {*n* ∈ *NODES_REMOVED* ∧ *n* ∈ *PRESENT*
PROPOSAL_NOT_PENDING: *PROPOSAL_CYCLE_#* ∉ *PROPOSALS_PENDING*
INSUFFICIENT_TRUSTEE_2_SIGNATURES: ¬ *check-signatured*
(*ANNOUNCEMENT_DEFINITION*, *trustee-2-publics*, *trustee-2-quorum*)
INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = 2 ∧
¬{let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT*, {∀*n*:**node-id**
{ if *n* ∈ *PRESENT* then *NODE_PUBLICS*[*n*]}}, *MAJORITY*))
INSTANT_NOT_SIGNATORY: ¬ ∃ *K*:**public-key** {*k* ∈ *ALL_OWN_NODE_PUBLICS* ∧
check-signature (*PROPOSAL*, *k*)}
TOO_EARLY: *now* − *LATEST_TIMESTAMP* < *cooling-off-interval*
Effects:
if ¬ *SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign* (*ANNOUNCEMENT*, *NODE_PRIVATE*), *PHASE* = 2}
else {
PROPOSAL_CYCLE_# ∉ *PROPOSALS_PENDING*,
CYCLE = '*CYCLE*'+1,
PHASE = 1,
COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),

$NODES\_IN\_USE$ = '$NODES\_IN\_USE$' – $NODES\_TO\_REMOVE$,
$USED\_NODE\_IDS$ = '$USED\_NODE\_IDS$' ∪ $NODES\_TO\_REMOVE$,
$APPLIED$ = '$APPLIED$' – $NODES\_TO\_REMOVE$,
$\forall n$:**node-id** { if $n \in NODES\_TO\_REMOVE$ then
{$PARTIAL\_KEYS[n]$ = *erased*,
$\forall i$:**integer** { if $1 \leq i \leq SUB-PARTIALS\_REMAINING[n]$ then
{$SUB-PARTIALS[n][i]$ = *erased*}}}}
if $OWN\_NODE \in NODES\_TO\_REMOVE$ then *suicide* (*0*)}

Trustee 1 *O*-functions

The *PROPOSE* and *CANCEL_PROPOSAL* functions were discussed above as they cross over the boundary between trustee level 1 and trustee level 2. The remaining functions covered in this section are illustrated in Figure 1.

The *APPLY* function takes any suitably certified node in a "veteran" state (i.e. a node that has been present before), and makes it "applied," that is dedicates it to a particular application and disqualifies it from being the replacing node in a restart. The *CHANGE_PRESENT* function transfers nodes between the present and participated states, and also may change the current majority. The *RESTART* function was touched on in Chapter IV, and is simply a way for a replacing node to resume the application processing of the disabled replaced node. The *PARTICIPATE* function can be used to transfer a single node from some state outside the participated state to the participated state, an effect which is usually achieved by a key change.

The remaining two functions are related. First, the *CREATE_KEYS* function is called and results in each node forming a new set of keys, and outputting the appropriate public keys. These public keys are then used in conjunction with un-synchronized *O*-functions, described in the following section, to distribute new partial and sub-partial keys among the nodes hoping to participate in the key change. Other synchronized *O*-functions may be taking place while these

keys are exchanged. Finally, during some later cycle, the *CHANGE_KEYS* O-function is called. It defines the new set of participated nodes and causes all business of the network to be conducted under the new keys.

Again, there are common exceptions and parts of the effects which provide synchronization. The function definitions are as follows:

PROPOSE: O-function

Input:

ANNOUNCEMENT_DEFINITION:**signatured**
<*PROPOSAL_DEFINITION*:
*<*NODE_CERTIFIED*:**node-id**,
NODE_KEY:**public-key**,
APPLICATION_KEY:**public-key**,
NODES_RESTARTABLE:set of **node-id**>,
*<*NEW_MINIMUM_QUORUM*:**integer**,
NEW_MINIMUM_MARGIN:**integer**,
NEW_SUICIDE_INTERVAL:**integer**>,
*<*NODES_TO_REMOVE*:set of **node-id**>,
KIND_OF_PROPOSAL:**proposal-kind**,
CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
propose = :**action-kind**>,

Output:

PROPOSAL:**signed**
<*PROPOSAL_DEFINITION*:,
LATEST_TIMESTAMP:**time**,
'*CYCLE*' = *CYCLE_#*:**integer**,
KIND_OF_PROPOSAL:**proposal-kind**>

Comment: A Signed copy of a definition of the proposed action, *PROPOSAL*, is output which includes the latest single timestamp of the quorum of nodes signing the announcement.

Exceptions:

INSUFFICIENT_TRUSTEE_1_SIGNATURES: ¬*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: $PHASE = 2 \wedge$
¬(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT_DEFINITION*, {$\forall n$:**node-id**
{ if $n \in PRESENT$ then $NODE\_PUBLICS[n]$}}, *MAJORITY*))

Effects:

if ¬*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), *PHASE* = *2*}
else {
CYCLE = '*CYCLE*'+*1*,
PHASE = *1*,
COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),
CYCLE_# ∈ *PROPOSALS_PENDING*,
LATEST_TIMESTAMP = *latest-signature*(*ANNOUNCEMENT_DEFINITION*),
sign(*PROPOSAL*, *NODE_PRIVATE*)}

CANCEL_PROPOSAL: *O*-function
Input:
ANNOUNCEMENT_DEFINITION:**signatured**
<*PROPOSALS_TO_CANCEL*:set of **integer**,
CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
cancel = :**action-kind**>
Comment: The *PROPOSALS_TO_CANCEL* are removed from *PROPOSALS_PENDING* and therefore can no longer be used.
Exceptions:
BAD_PROPOSALS: ¬*PROPOSALS_TO_CANCEL* ⊆ *PROPOSALS_PENDING*
INSUFFICIENT_TRUSTEE_1_SIGNATURES: ¬*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)
INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = *2* ∧
¬(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT_DEFINITION*, {∀*n*:**node-id**
{ if *n* ∈ *PRESENT* then *NODE_PUBLICS*[*n*]}}, *MAJORITY*))
Effects:
if ¬*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), *PHASE* = *2*}
else {
CYCLE = '*CYCLE*'+*1*,
PHASE = *1*,
COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),
PROPOSALS_PENDING = *PROPOSALS_PENDING* − *PROPOSALS_TO_CANCEL*}

APPLY: *O*-function

Input:

ANNOUNCEMENT_DEFINITION:**signatured**

<*NODES_TO_APPLY*:set of **node-id**,

CYCLE = *CYCLE_#*:**integer**,

COMPRESSED_HISTORY = :**integer**,

apply = :**action-kind**>

Comment: The identified node(s) are added to *APPLIED*, and all their certification is removed. They expunge their own set of trustee partials. The subject nodes can now adopt an application, and can no longer be used as the replacing node in a restart.

Exceptions:

BAD_NODES: $\neg$ *NODES_TO_APPLY* $\subseteq$ *NODES_IN_USE*

ALREADY_APPLIED: $\exists n$:**node-id** {$n \in$ *NODES_TO_APPLY* $\wedge$ $n \in$ *APPLIED*}

INADEQUATE_CERTIFICATION: $\neg \exists n$:**node-id** {$n \in$ *NODES_TO_APPLY* $\wedge$
$n \notin$ *CERTIFICATION*[n]

INSUFFICIENT_TRUSTEE_1_SIGNATURES: $\neg$*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = 2 $\wedge$
$\neg$(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT_DEFINITION*, {$\forall n$:**node-id**
{ if $n \in$ *PRESENT* then *NODE_PUBLICS*[n]}}, *MAJORITY*))

Effects:

if $\neg$*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), *PHASE* = 2}

else {

CYCLE = '*CYCLE*'+1,

PHASE = 1,

COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),

$\forall a$:**node-id** { if $a \in$ *NODES_TO_APPLY* then *CERTIFICATION*[a] = *empty* },

if *OWN_NODE* $\in$ *NODES_TO_APPLY* then $\forall n$:**node-id**
{ if $n \in$ *CERTIFICATION*[*OWN_NODE*] then
OWN_TRUSTEE_PARTIALS[n] = *erased* },

APPLIED = '*APPLIED*' $\cup$ *NODES_TO_APPLY*}

CHANGE_PRESENT: *O*-function

Input:

ANNOUNCEMENT_DEFINITION:**signatured**

<*NODES_TO_BECOME_PRESENT*:set of **node-id**,

NODES_TO_BECOME_ABSENT:set of **node-id**,

NEW_MAJORITY:**integer**,

CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
change-presents = :**action-kind**>
MINIMUM_PARTIALS_RECEIVED:**signatured**
<*KEY_CREATION_#* = :**integer**,
NODES_RECEIVED_FROM:set of **node-id**,
partials-received = :**transfer-kind**>

Output:

SUB–PARTIALS_RELEASED*:set of **partial-key

Comment: The nodes to be made present are made present, the nodes to be made absent are made absent, and the majority assumes the new value provided. The new configuration must be compatible with the *MINIMUM_MARGIN*, and the *QUORUM*. The *MINIMUM_PARTIALS_RECEIVED* signed by the *NODES_TO_BECOME_PRESENT* ensure that the nodes made present have received all the partial keys they may require. If the *NEW_MAJORITY* is less than the current quorum, but not less than the minimum quorum, then sub-partials are publicly released so that the effective quorum is lowered to the *NEW_MAJORITY*.

Exceptions:

NEW_MAJORITY_TOO_SMALL: $NEW\_MAJORITY < MINIMUM\_QUORUM$

NEW_MAJORITY_TOO_BIG: $NEW\_MAJORITY >$ (let $NEW\_NODE\_COUNT =$
$cardinality(PRESENT) + cardinality(NODES\_TO\_BECOME\_PRESENT) -$
$cardinality(NODES\_TO\_BECOME\_ABSENT)$)

NEW_MAJORITY_TOO_SMALL: $MINIMUM\_QUORUM > NEW\_MAJORITY$

INSUFFICIENT_MARGIN: $NEW\_NODE\_COUNT >$
$(NEW\_MAJORITY \times 2) - MINIMUM\_MARGIN$

NOT_ABSENT: $\neg NODES\_TO\_BECOME\_PRESENT \subseteq ABSENT$

NOT_PRESENT: $\neg NODES\_TO\_BECOME\_ABSENT \subseteq PRESENT$

INSUFFICIENT_MINIMUM_PARTIALS_RECEIVED_FROM_SIGNATURES:
$\neg$*check-signatured*(*MINIMUM_PARTIALS_RECEIVED*,
$NODE\_PUBLICS[(PRESENT \cup NODES\_TO\_BECOME\_PRESENT) -$
$NODES\_TO\_BECOME\_ABSENT]$,
NEW_NODE_COUNT)

INSUFFICIENT_MINIMUM_PARTIALS_RECEIVED_FROM: $\exists n$:**node-id**
$\{n \in NODES\_TO\_BECOME\_PRESENT \wedge$
$n \notin NODES\_RECEIVED\_FROM\}$

INSUFFICIENT_SUB–PARTIALS:
$\exists n$:**node-id** $\{n \in NODES\_IN\_USE \wedge \exists i$:**integer**
$\{LAST\_CHANGES[n] = i \wedge$
$SUB-PARTIALS\_REMAINING[n] < QUORUMS[i] - NEW\_MAJORITY\}\}$

INSUFFICIENT_TRUSTEE_1_SIGNATURES: $\neg$*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: $PHASE = 2 \wedge$
$\neg$(let $SIGNATURES\_OF\_MAJORITY\_OF\_PRESENTS =$
check-signatured(*ANNOUNCEMENT_DEFINITION*, $\{\forall n$:**node-id**
$\{$ if $n \in PRESENT$ then $NODE\_PUBLICS[n]\}\}$, *MAJORITY*))

Effects:

if ~*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), *PHASE* = *2*}
else {
CYCLE = '*CYCLE*'+*1*,
PHASE = *1*,
COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),
PRESENT = ('*PRESENT*'–*NODES_TO_BECOME_ABSENT*) ∪
NODES_TO_BECOME_PRESENT,
MAJORITY = *NEW_MAJORITY*}
if *MAJORITY* < *QUORUM* then ∀*n*:**node-id**
{ if *n* ∈ *NODES_IN_USE* ∧ (let *NS_QUORUM* =
QUORUMS[*LAST_CHANGES*[*n*]]) > *NEW_MAJORITY* then
{{*SUB–PARTIALS_REMAINING*[*n*] =
'*SUB–PARTIALS_REMAINING*'[*n*]–
NS_QUORUM–NEW_MAJORITY},
{∀*i*:**integer** { if *NS_QUORUM* < *i* ≤ *NEW_MAJORITY* then
SUB–PARTIALS[*n*]
['*SUB–PARTIALS_REMAINING*'[*n*]–*i*–
NEW_MAJORITY] ∈ *SUB–PARTIALS_RELEASED*}}}}

RESTART:*O*-function

Input:

ANNOUNCEMENT_DEFINITION:**signatured**
<*REPLACED_NODE*:**node-id**,
REPLACING_NODE:**node-id**,
CHECKPOINT: (see *ISSUE_CHECKPOINT*),
CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
restart = :**action-kind**>

Output:

PARTIAL_FOR_ASSUME_APPLICATION*:signed**
<*REPLACED_NODE*:**node-id**,
REPLACING_NODE:**node-id**,
PARTIAL_SUPPLIED:**sealed**<**partial-key**>,
RESTART_to_ASSUME_APPLICATION = :**transfer-kind**>

Comment: The replaced node, which must be applied and not present, is in effect *REMOVE_NODES*ed. The replacing node must be certified to replace the replaced node, and it becomes applied. The replacing node is supplied with partials for the replaced node. These are used in *ASSUME_APPLICATION* to recover the replaced node's application data and messages sent after the last checkpoint.

Exceptions:

BAD_REPLACED_NODE: *REPLACED_NODE* ∈ *PRESENT* ∨
REPLACED_NODE ∉ *NODES_IN_USE*

BAD_REPLACEMENT_NODE: *REPLACING_NODE* ∉ *PARTICIPATED*
REPLACING_NODE ∈ *APPLIED*

INADEQUATE_CERTIFICATION: *REPLACED_NODE* ∉
CERTIFICATION[*REPLACING_NODE*]

INSUFFICIENT_TRUSTEE_1_SIGNATURES: ¬*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = 2 ∧
¬(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT_DEFINITION*, {∀*n*:**node-id**
{ if *n* ∈ *PRESENT* then *NODE_PUBLICS*[*n*]}}, *MAJORITY*))

Effects:

if ¬*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), *PHASE* = 2}

else {

CYCLE = '*CYCLE*'+1,

PHASE = 1,

COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),

REPLACED_NODE ∉ *NODES_IN_USE*,

REPLACED_NODE ∈ *USED_NODE_IDS*,

APPLICATION_PUBLICS[*REPLACING_NODE*] =
APPLICATION_PUBLICS[*REPLACED_NODE*],

REPLACING_NODE ∈ *APPLIED*,

REPLACED_NODE ∉ *APPLIED*,

PARTIAL_KEYS[*REPLACED_NODE*] = *erased*,

∀*i*:**integer** { if 1 ≤ *i* ≤ *SUB−PARTIALS_REMAINING*[*n*] then
{*SUB−PARTIALS*[*REPLACED_NODE*][*i*] = *erased*},

PARTIAL_SUPPLIED = '*PARTIAL_KEYS*'[*REPLACED_NODE*],

seal(*PARTIAL_SUPPLIED*, *NODE_PUBLICS*[*REPLACING_NODE*]),

sign(*PARTIAL_FOR_ASSUME_APPLICATION*, *NODE_PRIVATE*),

if *OWN_NODE* = *REPLACING_NODE* then ∀*n*:**node-id**
{ if *n* ∈ *CERTIFICATION*[*REPLACING_NODE*] then
OWN_TRUSTEE_PARTIALS[*n*] = *erased*}},

if *OWN_NODE* = *REPLACED_NODE* then *suicide*(0)}

PARTICIPATE:*O*-function

Input:

ANNOUNCEMENT_DEFINITION:**signatured**

<*NODE_PARTICIPATED*:**node-id**,

NODES_RECEIVED_FROM:set of **node-id**

PARTIALS_ALREADY_RECEIVED:**signatured**

<*KEY_CREATION_#* = :**integer**,

NODES_RECEIVED_FROM:set of **node-id**,

partials-received = :**transfer-kind**>,

CYCLE = *CYCLE_#*:**integer**,

COMPRESSED_HISTORY = :**integer**,

participate = :**action-kind**>

Output:

SUB-PARTIALS_SUPPLIED*:signed**

<*CYCLE* = *CYCLE_#*:**integer**,

SUB-PARTIALS:**sealed**<:table[**node-id**] of **partial-key**>,

PARTICIPATE_to_NEW_PARTICIPANT_RECEIVE = :
transfer-kind>

**PARTIALS_AND_SUB-PARTIALS*:table[*PARTICIPATED*] of

PARTIAL_AND_SUB-PARTIALS:**signed**

<*RECIPIENT*:**node-id**,

PARTIAL:**sealed**<**partial-key**>,

NUMBER_OF_SUB-PARTIALS:**integer**,

SUB-PARTIALS:**sealed**<table[**integer**] of **partial-key**>,

PARTICIPATE_to_RECEIVE_NEW_PARTICIPANT = :
transfer-kind>

Comment: Participated nodes each supply the node to be participated with sub-partials of every node for which the node to be participated is missing partial keys. The node to be participated issues partials and sub-partials for itself to all the participated nodes, just as in issue-partials. Two unsynchronized *O*-functions are allowed: *RECEIVE_NEW_PARTICIPANT* for the participated nodes to pick up their partials and sub-partials (not as new), and *NEW_PARTICIPANT_RECEIVE* for the entering node to pick up a set of sub-partials.

Exceptions:

BAD_NODE_PARTICIPATED: $NODE\_PARTICIPATED \in PARTICIPATED \vee$
$NODE\_PARTICIPATED \notin NODES\_IN\_USE$

INSUFFICIENT_SUB-PARTIALS: $\exists n$:**node-id**$\{n \in NODES\_IN\_USE \wedge$
$LAST\_CHANGES[NODE\_PARTICIPATED] < LAST\_CHANGES[n] \wedge$
$SUB\text{-}PARTIALS\_REMAINING[n] < 1\}$

BAD_ALREADY_RECEIVED_FROM_SIGNATURES: $\neg$*check-signatured*
(*PARTIALS_ALREADY_RECEIVED*,
$NODE\_PUBLICS[NODES\_RECEIVED\_FROM] \cup$
$NODE\_PUBLICS[OWN\_NODE]$,
cardinality(*NODES_RECEIVED_FROM*+1))

INSUFFICIENT_TRUSTEE_1_SIGNATURES: ¬*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = 2 ∧
¬(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT_DEFINITION*, {∀*n*:**node-id**
{ if *n* ∈ *PRESENT* then *NODE_PUBLICS*[*n*]}}, *MAJORITY*))

Effects:

if ¬*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), *PHASE* = 2}
else {
CYCLE = '*CYCLE*'+1,
PHASE = 1,
COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),
SUB−PARTIALS_REMAINING[*NODE_PARTICIPATED*] =
NUMBER_OF_SUB−PARTIALS,
if ∃*n*:**node-id**{*n* ∈ *NODES_IN_USE* ∧
LAST_CHANGES[*NODE_PARTICIPATED*] < *LAST_CHANGES*[*n*] then
SUB−PARTIALS_REMAINING[*n*] =
'*SUB−PARTIALS_REMAINING*'[*n*]−1}
if *NODE_PARTICIPATED* ≠ *OWN_NODE* then
{∀
p:**node-id** { if *p* ∈ *PARTICIPATED* ∧ *p* ∉ *NODES_RECEIVED_FROM* then
{*SUB−PARTIALS*[*p*] =
SUB−PARTIALS[*p*]['*SUB−PARTIALS_REMAINING*'[*p*]],
seal(*SUB−PARTIALS*[*p*], *NODE_PUBLICS*[*NODE_PARTICIPATED*])},
sign(*SUB−PARTIALS_SUPPLIED*, *NODE_PRIVATE*)},
else
{∀
p:**node-id** { if *p* ∈ *PARTICIPATED* ∧ *p* ∉ *NODES_RECEIVED_FROM* then
{with *PARTIALS_AND_SUBPARTIALS*[*p*]
{*RECIPIENT* = *p*,
PARTIAL = *form-partial*(*p*, *PARTIAL_SEED*,
APPLICATION_PRIVATE,
QUORUM),
seal(*PARTIAL*, *NODE_PUBLICS*[*p*]),
∀*i*:**integer** { if 1 ≤ *i* ≤ *NUMBER_OF_SUBPARTIALS* then
{*SUB−PARTIALS*[*i*] = *form-partial*(
p, *PARTIAL_SEED*,
form-partial(*i*, *PARTIAL_SEED*,
APPLICATION_PRIVATE,
QUORUM),
QUORUM)
seal(*SUB−PARTIALS*, *NODE_PUBLICS*[*p*]])}}
sign(*PARTIAL_AND_SUB−PARTIALS*, *NODE_PRIVATE*)},

suicide ((*earliest-signature* (*ANNOUNCEMENT_DEFINITION*) +
SUICIDE_INTERVAL) − *now*())}}}

CREATE_KEYS: *O*-function

Input:

ANNOUNCEMENT_DEFINITION:**signatured**
<*NEW_QUORUM*:**integer**,
NEW_SUB−PARTIALS_NEEDED:table[*NODES_IN_USE*] of **integer**,
CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
create-keys = :**action-kind**>

Output:

NEW_KEYS:**signed**
<*KEY_CREATION_#* = :**integer**,
NEW_APPLICATION_PUBLIC:**public-key**,
NEW_NODE_PUBLIC:**public-key**,
CREATE_KEYS_to_ISSUE_NEW_PARTIALS&CHANGE_KEYS = :
transfer-kind>
EXTENDERS:table[**integer**] of
EXTENDER:**signed**
<*KEY_CREATION_#* = :**integer**,
INDEX:**integer**,
EXTENSION:**sealed**<table[**integer**] of **partial-key**>,
CREATE_KEYS_to_NEW_PARTICIPANT_RECEIVE = :
transfer-kind>

Comment: New node and application keys are created. Also a new seed for the new partial keys, which will use the new quorum, is created. The initial number of sub-partials needed for each node is recorded. New publics are output. The issue new partials and receive new partials unsynchronized actions are allowed. This action may occur more than once to change the new quorum, even though no change to new keys has occurred. The set of nodes the instant node would allow to become participated in a *CHANGE_KEYS* is emptied. If a node is to become participated but lacks partials for some other node which is not going to be participated, then the first node must be the subject of a *PARTICIPATE* before a *CHANGE_KEYS*.

Exceptions:

NEW_QUORUM_TOO_SMALL: *NEW_QUORUM* < *MINIMUM_QUORUM*

INSUFFICIENT_TRUSTEE_1_SIGNATURES: ¬*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: *PHASE* = 2 ∧
¬(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT_DEFINITION*, {∀*n*:**node-id**
{ if *n* ∈ *PRESENT* then *NODE_PUBLICS*[*n*]}, *MAJORITY*))

Effects:

if ¬*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), *PHASE* = *2*}
else {
CYCLE = 'CYCLE'+*1*,
PHASE = *1*,
COMPRESSED_HISTORY =
compress('*CONSENSUS_V-FUNCTIONS*'),
KEY_CREATION_# = '*KEY_CREATION_#*'+*1*,
QUORUMS[*LAST_CHANGE* + *1*] = *NEW_QUORUM*,
PARTIALS_RECEIVED_FROM = *empty*,
APPLICATION_SEED = *create-seed*(),
NEW_APPLICATION_PRIVATE = *create-private*(*APPLICATION_SEED*),
NEW_APPLICATION_PUBLIC = *create-public*(*APPLICATION_SEED*),
NODE_SEED = *create-seed*(),
NEW_NODE_PRIVATE = *create-private*(*NODE_SEED*),
NEW_NODE_PUBLIC = *create-public*(*NODE_SEED*),
PARTIAL_SEED = *create-seed*()},
NEW_SUB−PARTIALS_REMAINING = *NEW_SUB−PARTIALS_NEEDED*,
∀*i*:**integer**
{ if *1* < *i* ≤ *NEW_SUB−PARTIALS_REMAINING*[*OWN_NODE*] then
∀*J*:**integer** { if *1* < *J* < *i* then
{with *EXTENDERS*[*i*]
{*INDEX* = *i*,
EXTENSION[*J*] = *form-partial*(
i, *PARTIAL_SEED*, *form-partial*
(*J*, *PARTIAL_SEED*,
NEW_APPLICATION_PRIVATE,
QUORUMS[*LAST_CHANGE* + *1*]),
QUORUMS[*LAST_CHANGE* + *1*])}}}
seal(*EXTENSION*, *create-public*(*form-partial*
(*i*, *PARTIAL_SEED*, *NEW_APPLICATION_PRIVATE*,
QUORUMS[*LAST_CHANGE* + *1*]))),
sign(*EXTENDER*, *NEW_NODE_PRIVATE*)}

CHANGE_KEYS: *O*-function
Input:
ANNOUNCEMENT_DEFINITION:**signatured**
<*NODES_PARTICIPATING*:set of **node-id**,
EVERY_PARTICIPANTS_NEW_KEYS:table[**node-id**] of
NEW_KEYS_FROM_CREATE_KEYS:**signed**

<KEY_CREATION_# = :**integer**,
NEW_APPLICATION_PUBLIC:**public-key**,
NEW_NODE_PUBLIC:**public-key**,
CREATE_KEYS_to_ISSUE_NEW_PARTIALS&CHANGE_KEYS = :
transfer-kind>
CYCLE = *CYCLE_#*:**integer**,
COMPRESSED_HISTORY = :**integer**,
change-keys = :**action-kind**>
MINIMUM_PARTIALS_RECEIVED:**signatured**
<*KEY_CREATION_#* = :**integer**,
NODES_RECEIVED_FROM:set of **node-id**,
partials-received = :**transfer-kind**>

Comment: The new keys, partials and sub-partials for all the included nodes are changed to their new values that have previously been supplied. (Conflict between the supplied publics and any publics already received are ignored because this causes no real security problem, and if the *O*-function were blocked by a conflict, a single node could deadlock the system.) The new keys, partials, and sub-partials for all un-included nodes, except the present node, are erased. The set of participated nodes is changed to the included nodes.

Exceptions:

INSUFFICIENT_PARTICIPATION: $\neg PRESENT \subseteq NODES\_PARTICIPATING$

BAD_PARTICIPANTS: $\neg NODES\_PARTICIPATING \subseteq NODES\_IN\_USE$

INVALID_NEW_KEYS: $\exists n$:**node-id** $\{n \in NODES\_PARTICIPATING \wedge$
$\neg$*check-signature*(*EVERY_PARTICIPANTS_NEW_KEYS*$[n]$,
NODE_PUBLICS$[n]$)

PARTICIPANTS_LACK_NON-PARTICIPANTS_PARTIALS:$\exists p, n$:**node-id**
$\{p \in NODES\_PARTICPATING \wedge$
$n \in (NODES\_IN\_USE - NODES\_PARTICIPATING) \wedge$
$LAST\_CHANGES[p] < LAST\_CHANGES[n]\}$

INSUFFICIENT_MINIMUM_PARTIALS_RECEIVED_FROM_SIGNATURES:
$\neg$*check-signatured*(*MINIMUM_PARTIALS_RECEIVED*,
NODE_PUBLICS[*NODES_PARTICIPATING*],
cardinality(*NODES_PARTICIPATING*))

INSUFFICIENT_MINIMUM_PARTIALS_RECEIVED_FROM:
$\neg NODES\_PARTICIPATING \subseteq NODES\_RECEIVED\_FROM$

NEW_QUORUM_TOO_BIG: $QUORUMS[LAST\_CHANGE + 1] > MAJORITY$

INSUFFICIENT_TRUSTEE_1_SIGNATURES: $\neg$*check-signatured*
(*ANNOUNCEMENT*, *trustee-1-publics*, *trustee-1-quorum*)

INSTANT_ALREADY_SIGNED_ANNOUNCEMENT: $PHASE = 2 \wedge$
$\neg$(let *SIGNATURES_OF_MAJORITY_OF_PRESENTS* =
check-signatured(*ANNOUNCEMENT_DEFINITION*, $\{\forall n$:**node-id**
{ if $n \in PRESENT$ then *NODE_PUBLICS*$[n]\}\}$, *MAJORITY*))

Effects:

if $\neg$*SIGNATURES_OF_MAJORITY_OF_PRESENTS* then
{*sign*(*ANNOUNCEMENT_DEFINITION*), $PHASE = 2\}$

else {
CYCLE = '*CYCLE*'+1,
PHASE = 1,
COMPRESSED_HISTORY =
    compress('*CONSENSUS_V-FUNCTIONS*'),
PARTICIPATED = *NODES_PARTICIPATING*,
QUORUM = *QUORUMS*[*LAST_CHANGE* + 1],
SUB-PARTIALS_REMAINING = *NEW_SUB-PARTIALS_REMAINING*,
NODE_PUBLICS[*OWN_NODE*] ∈ *ALL_OWN_NODE_PUBLICS*,
PARTIALS_RECEIVED_FROM = *NODES_PARTICIPATING*,
∀*p*:**node-id** { if *p* ∈ *PARTICIPATED* then
    {with *EVERY_PARTICIPANTS_NEW_KEYS*[*p*]
        {*APPLICATION_PUBLICS*[*p*] = *NEW_APPLICATION_PUBLIC*},
        NODE_PUBLICS[*p*] = *NEW_NODE_PUBLIC*},
    PARTIAL_KEYS[*p*] = *NEW_PARTIAL_KEYS*[*p*],
    ∀*i*:**integer** { if 1 ≤ *i* ≤ *NEW_SUB-PARTIALS_REMAINING*[*p*] then
        SUB-PARTIALS[*p*][*i*] = *NEW_SUB-PARTIALS*[*p*][*i*]}},
suicide ((*earliest-signature* (*ANNOUNCEMENT_DEFINITION*) +
    SUICIDE_INTERVAL) - *now*())}

§7 Un-Synchronized *O*-functions

Presented are the remaining *O*-functions, which support the *O*-functions of the previous section and allow release of information.

The previous section was concerned with synchronized *O*-functions, which are designed in such a way that every node will accept only the same sequence of calls and in the same order. The present section is concerned with the other *O*-functions: those which can be invoked in many possible orders. The fact that they can be used in a less structured way than those previously discussed does not mean that these *O*-functions are an invitation to chaos. On the contrary, some of these *O*-functions provide increased reliability and robustness of the network even in spite of the trustees. Others of these *O*-functions have no effect on a node's state, and are merely used to obtain signed and possibly sealed data about the node's state. Yet others are tied directly into the synchronized *O*-

functions, and merely act as extensions of these *O*-functions to allow additional rounds of information exchange.

Synchronized *O*-function Support

This subsection defines five un-synchronized *O*-functions. The first two are used between the initial *CREATE_KEYS* and the closing *CHANGE_KEYS*, as described in the previous section. The first of these, *ISSUE_NEW_PARTIALS*, takes as input the new public keys of a node released during *CREATE_KEYS* and outputs partial keys and sub-partial keys sealed with the new public key received. The second *O*-function, *RECEIVE_NEW_PARTIALS*, takes as input the output of this first *O*-function created by another node and simply records the partials and sub-partials after unsealing with its new private key.

A second pair of *O*-functions serves a similar purpose, but is used following a *PARTICIPATE* *O*-function call. One *O*-function, *RECEIVE_NEW_PARTICIPANT*, is used by all but the node to be participated, and simply records the public, partial, and sub-partial keys released by the subject node during the *PARTICIPATE*. The other *O*-function of the pair, *NEW_PARTICIPANT_RECEIVE*, is used by the subject node to collect the sub-partials and extenders provided it by the non-subjects during the *PARTICIPATE*. The fifth and final *O*-function, *ASSUME_APPLICATION*, allows the replacing node of a restart to assume the application key of the replaced node.

The following are the unsynchronized supporting *O*-functions:

ISSUE_NEW_PARTIALS: *O*-function

Input:

SUPPLIER:**node-id**,

NEW_PUBLICS:**signed**

<*KEY_CREATION_#* = :**integer**,

SUPPLIER_NEW_NODE_PUBLIC:**public-key**,

SUPPLIER_NEW_APPLICATION_PUBLIC:**public-key**,

CREATE_KEYS_to_ISSUE_NEW_PARTIALS&CHANGE_KEYS = :
transfer-kind>

Output:

PARTIAL_AND_SUB-PARTIALS:**signed**

<*KEY_CREATION_#*=:**integer**,

SUPPLIER:**node-id**,

PARTIAL:**sealed**<:**partial-key**>,

SUB-PARTIALS:**sealed**<:table[**integer**] of **partial-key**>,

ISSUE_NEW_PARTIALS_to_RECEIVE_NEW_PARTIALS = :
transfer-kind>

Comment: The supplied new public application key is used to seal the partial and the number of sub-partials for each node established in *CREATE_KEYS*.

Exceptions:

INVALID_SUPPLIER: *SUPPLIER* ∉ *NODES_IN_USE*,

INVALID_SUPPLIER_SIGNATURE: ¬*check-signature*(*NEW_PUBLICS*,
NODE_PUBLICS(*SUPPLIER*)),

Effects:

PARTIAL = *form-partial*(*SUPPLIER*, *PARTIAL_SEED*,
NEW_APPLICATION_PRIVATE, *QUORUMS*[*LAST_CHANGE* + *1*]),

seal(*PARTIAL*, *SUPPLIER_NEW_NODE_PUBLIC*),

∀*i*:**integer**
{ if *1* ≤ *i* ≤ *NEW_SUB-PARTIALS_REMAINING*[*OWN_NODE*] then
{*SUB-PARTIALS*[*i*] = *form-partial*
(*SUPPLIER*, *PARTIAL_SEED*,
form-partial(*i*, *PARTIAL_SEED*,
NEW_APPLICATION_PRIVATE,
QUORUMS[*LAST_CHANGE* + *1*]),
QUORUMS[*LAST_CHANGE* + *1*])},

seal(*SUB-PARTIALS*, *SUPPLIER_NEW_NODE_PUBLIC*),

sign(*PARTIAL_AND_SUB-PARTIALS*, *NODE_PRIVATE*)

RECEIVE_NEW_PARTIALS: *O*-function

Input:

SUPPLIER:**node-id**,

PARTIAL_AND_SUB-PARTIALS:**signed**

<*KEY_CREATION_#* = :**integer**,

OWN_NODE:**node-id**,

PARTIAL:**sealed**<:**partial-key**>,

SUB-PARTIALS:**sealed**<:table[**integer**] of **partial-key**>,

ISSUE_NEW_PARTIALS_to_RECEIVE_NEW_PARTIALS = :
transfer-kind>

Comment: The new partials and sub-partials output by *ISSUE_NEW_PARTIALS* are recorded.

Exceptions:

INVALID_SUPPLIER:*SUPPLIER* ∉ *NODES_IN_USE*

INVALID_SUPPLIER_SIGNATURE: *check-signature*
(*PARTIAL_AND_SUB-PARTIALS*, *NODE_PUBLICS*[*SUPPLIER*])

Effects:

unseal(*PARTIAL*, *NEW_APPLICATION_PRIVATE*),

NEW_PARTIAL_KEYS[*SUPPLIER*] = *PARTIAL*,

unseal(*SUB-PARTIALS*, *NEW_APPLICATION_PRIVATE*)

∀*i*:**integer** { if $1 \le i \le$ *NEW_SUB-PARTIALS_REMAINING*[*SUPPLIER*] then
NEW_SUB-PARTIALS[*SUPPLIER*][*i*] = *SUB-PARTIALS*[*i*]},

SUPPLIER ∈ *PARTIALS_RECEIVED_FROM*,

RECEIVE_NEW_PARTICIPANT: *O*-function

Input:

NODE_BECOMING_PARTICIPATED:**node-id**

PARTIAL_AND_SUB-PARTIALS:**signed**

<*OWN_NODE* = *RECIPIENT*:**node-id**,

PARTIAL:**sealed**<:**partial-key**>,

NUMBER_OF_SUB-PARTIALS:**integer**,

SUB-PARTIALS:**sealed**<:table[**integer**] of **partial-key**>,

PARTICIPATE_to_RECEIVE_NEW_PARTICIPANT = :
transfer-kind>

Comment: The participated nodes making an additional node participated are allowed to use this *O*-function to record the partials and sub-partials issued to them by the entering node during the *BECOME_PARTICIPATED*.

Exceptions:

BAD_SIGNATURE: *check-signature*
(*PARTIAL_AND_SUB-PARTIALS*,
NODE_PUBLICS[*NODE_BECOMING_PARTICIPATED*])

Effects:

unseal(*PARTIAL*, *APPLICATION_PRIVATE*),

PARTIAL_KEYS[*NODE_BECOMING_PARTICIPATED*] = *PARTIAL*,

{*unseal*(*SUB-PARTIALS*, *APPLICATION_PRIVATE*)

$\forall i$\{ if $1 \leq i \leq$ *NUMBER_OF_SUB-PARTIALS* then

SUB-PARTIALS[*NODE_BECOMING_PARTICIPATED*][i] =
SUB-PARTIALS[i]}},

NODE_BECOMING_PARTICIPATED $\in$ *PARTIALS_RECEIVED_FROM*,

NEW_PARTICIPANT_RECEIVE: *O*-function

Input:

SUB-PARTIAL_SUPPLIERS:set of **node-id**

ALL_SUB-PARTIALS_SUPPLIED:table[*SUB-PARTIAL_SUPPLIERS*] of

SUB-PARTIALS_SUPPLIED:**signed**

<*CYCLE* = *CYCLE_#*:**integer**,

SUB-PARTIALS:**sealed**<:table[**node-id**] of **partial-key**>,

PARTICIPATE_to_NEW_PARTICIPANT_RECEIVE = :
transfer-kind>

EXTENDERS:table[**node-id**] of

EXTENDER:**signed**

<*KEY_CREATION_#* = :**integer**,

INDEX:**integer**,

EXTENSION:**sealed**<:table[**integer**] of **partial-key**>,

CREATE_KEYS_to_NEW_PARTICIPANT_RECEIVE = :
transfer-kind>

Comment: The node which has become participated is allowed to use this *O*-function to obtain the sub-partials issued it by the quorum of participated nodes during a *PARTICIPATE*, and thereby obtain a full set of partials and also sub-partials. It is then able to list itself in its *PARTIALS_RECEIVED_FROM*, indicating it has received sufficient partials and allowing it to become present.

Exceptions:

NOT_ENOUGH_SUPPLIERS: *cardinality*(*SUB-PARTIAL_SUPPLIERS*) <
QUORUM

BAD_SUPPLIERS: $\neg$ *SUB-PARTIAL_SUPPLIERS* $\subseteq$ *PARTICIPATED*

BAD_SUPPLIER_SIGNATURE: $\exists n$:**node-id** {$n \in$ *SUB-PARTIAL_SUPPLIERS* $\wedge$
$\neg$ *check-signature* (*SUB-PARTIALS_SUPPLIED*[n], *NODE_PUBLICS*[n]}

BAD_EXTENDER_SIGNATURE: $\exists n$:**node-id**
{*LAST_CHANGES*[*OWN_NODE*] < *LAST_CHANGES*[n] $\wedge$
$\neg$ *check-signature* (*EXTENDERS*[n], *NODE_PUBLICS*[n])}

WRONG_EXTENDER: $\exists n$:**node-id**
{*LAST_CHANGES*[*OWN_NODE*] < *LAST_CHANGES*[n] $\wedge$
$\neg$ with *EXTENDERS*[n]

{*INDEX* ≠ *SUB–PARTIALS_REMAINING*[*n*]}}

Effects:

∀*s*:**node-id** { if *s* ∈ *SUB–PARTIAL_SUPPLIERS* then

with *ALL_SUB–PARTIALS_SUPPLIED*[*s*]

unseal(*SUB–PARTIALS*, *APPLICATION_PRIVATE*)},

∀*p*:**node-id**

{ if *LAST_CHANGES*[*OWN_NODE*] < *LAST_CHANGES*[*p*] then

PARTIAL_KEYS[*p*] = *merge-partials*(∀*s*:**node-id**

{ if *s* ∈ *SUB–PARTIAL_SUPPLIERS* then

with *SUB–PARTIAL_SUPPLIERS*[s]{*SUB–PARTIALS*[*p*]}}),

∀*p*:**node-id**

{ if *LAST_CHANGES*[*OWN_NODE*] < *LAST_CHANGES*[*p*] then

{with *EXTENDERS*[*p*]

{*unseal*(*EXTENSION*, *create-private*(*PARTIAL_KEYS*[*p*])),

{∀*i*:**integer**

{ if *1* ≤ *i* ≤ *SUB–PARTIALS_REMAINING*[*p*] then

SUB–PARTIALS[*p*][*i*] = *EXTENDER*[*i*]}}}},

OWN_NODE ∈ *PARTIALS_RECEIVED_FROM*

ASSUME_APPLICATION: *O*-function

Input:

PARTIAL_SUPPLIERS:set of **node-id**

PARTIALS_SUPPLIED:table[**node-id**] of

PARTIAL:**signed**

<*REPLACED_NODE*:**node-id**,

OWN_NODE = *REPLACING_NODE*:**node-id**,

PARTIAL_SUPPLIED:**sealed**<:**partial-key**>,

RESTART_to_ASSUME_APPLICATION = :**transfer-kind**>

CHECKPOINT: (see *ISSUE_CHECKPOINT*)

Comment: The replacing node of a restart is enabled to perform this operation, which involves receiving partials for the replaced node, and using them to obtain the saved application data from the checkpoint formed by the replaced node, and messages sent after the checkpoint was formed.

Exceptions:

BAD_SUPPLIERS: ¬ *PARTIAL_SUPPLIERS* ⊆ *NODES_IN_USE*

BAD_SUPPLIER_SIGNATURES: ∃*n*:**node-id** {*n* ∈ *PARTIAL_SUPPLIERS* ∧

{with *PARTIALS_SUPPLIED*[*n*]

¬*check-signature*(*PARTIAL*, *NODE_PUBLICS*[*n*])}

Effects:

∀*n*:**node-id** { if *n* ∈ *PARTIAL_SUPPLIERS* then

with *PARTIALS_SUPPLIED*[*n*]

unseal(*PARTIAL_SUPPLIED*, *APPLICATION_PRIVATE*)},

$\forall n$:**node-id** { if $n \in$ *PARTIAL_SUPPLIERS* then
APPLICATION_PUBLICS =
merge-partials (with *PARTIALS_SUPPLIED*[n]*PARTIAL_SUPPLIED*)}

Information Releasing *O*-functions

The last two *O*-functions are given in this subsection. Unlike the previous *O*-functions, they are not closely tied to any particular synchronized *O*-functions. They release information about the node's state, but do not alter its state. The *PARTIALS_RECEIVED* *O*-function allows a node to provide a signed statement about those nodes it has received current partial keys from. The *ISSUE_CHECKPOINT* *O*-function is unique in that it includes no input template, which means it does not check any signatures of input parameters, and thus can be freely called by anyone. This is appropriate because the output of this *O*-function provides, among other things, an authenticated snap-shot of the node's public state, which Chapter VII will show to be useful to those seeking to trust the network. Another use of checkpoints, that of saving enough of a node's state to make restart possible, was mentioned in Chapter IV. A further practical use of this *O*-function is to allow verification of the state certified into a node, which can allow new nodes to skip over a possibly long prefix of synchronized *O*-function calls.

The following are the information releasing un-synchronized *O*-functions:

PARTIALS_RECEIVED:*O*-function
Input:
MINIMUM_PARTIALS_RECEIVED:**signatured**
<*KEY_CREATION_#* = :**integer**,
NODES_RECEIVED_FROM:set of **node-id**,
partials-received = :**transfer-kind**>
Comment: The instant node adds its signature to the set of node ids input iff this set is a subset of the instant node's *PARTIALS_RECEIVED_FROM*.

Exceptions:

NOT_ALL_PARTIALS_RECEIVED: $\exists n$:**node-id**
$\{n \in$ *NODES_RECEIVED_FROM* $\wedge$
$n \notin$ *PARTIALS_RECEIVED_FROM* $\wedge$
$n \neq$ *OWN_NODE*$\}$ $\wedge$
KEY_CREATION_# $\neq 0$

Effects:

sign(*MINIMUM_PARTIALS_RECEIVED*, *NODE_PRIVATE*)

ISSUE_CHECKPOINT: *O*-function

Output:

CHECKPOINT:**signed**
<*INDIVIDUAL_V-FUNCTIONS* = : **any-type**,
CONSENSUS_V-FUNCTIONS = : **any-type**,
ALL_CURRENT_SECRETS:
sealed<*APPLICATION_SECRET_V-FUNCTIONS* = :
any-type>,
ALL_NEW_SECRETS: **sealed**
<*NEW_APPLICATION_SECRET_V-FUNCTIONS* = :
any-type>,
checkpoint: **transfer-kind**>

Comment: Causes the receiving node to output a signed copy of all its *INDIVIDUAL_V-FUNCTIONS* and *CONSENSUS_V-FUNCTIONS* state. A copy of its current secret state sealed with its current *APPLICATION_PUBLICS* and a copy of its new secret state sealed with *NEW_APPLICATION_PUBLIC*.

Effects:

seal(*ALL_CURRENT_SECRETS*, *APPLICATION_PUBLICS*),
seal(*ALL_NEW_SECRETS*, *NEW_APPLICATION_PUBLIC*),
sign(*CHECKPOINT*, *APPLICATION_PRIVATE*)

Chapter IV

Single Vault Systems

A simple single vault system is presented to introduce and illustrate some of the basic ideas of the proposed systems, and also to motivate and define the problems to be overcome by multiple vault systems.

When a certified vault is first constructed by the techniques presented in Chapter IX, a suitable public key and its inverse private key are chosen by a mechanism within the vault's protected interior, using a physically random process as discussed in Chapter II. The public key is then displayed outside the vault, on a special device certified for this purpose. As far as the world outside the vault is concerned, the possessor of the vault's private key is the vault: it can read sealed confidential messages sent to the vault, and it can make the vault's signature.

§1 Checkpoints & Restarts

Introduces the notions of encrypted checkpoints and the restarts they can allow trustees to perform.