The Design of a COTS Real-Time Distributed Security Kernel (Extended Version)

Miguel Correia
Paulo Veríssimo
Nuno Ferreira Neves

DI–FCUL TR–01–12

December 2001

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

The Design of a COTS Real-Time Distributed Security Kernel (Extended Version)

Miguel Correia   Paulo Veríssimo   Nuno Ferreira Neves
Department of Computer Science
University of Lisbon
Portugal

December 2001

Abstract

This technical report describes the design of a security kernel called TTCB, which has innovative features. Firstly, it is a distributed subsystem with its own secure network. Secondly, the TTCB is real-time, that is, a synchronous subsystem capable of timely behavior. These two characteristics together are uncommon in security kernels. Thirdly, the TTCB can be implemented using only COTS components.

We discuss essentially three things in this paper: (1) The TTCB is a simple component providing a small set of basic secure services. It aims at building a new style of protocols to achieve intrusion tolerance, which for the most part execute in insecure, arbitrary failure environments, and resort to the TTCB only in crucial parts of their operation. (2) Besides, the TTCB is a synchronous device supplying functions that may be an enabler of a new generation of timed secure protocols, until now known to be fragile due to attacks on timing assumptions. (3) Finally, we present a design methodology that establishes our hybrid failure assumptions in a well-founded manner. It helps us to achieve a robust design, despite using exclusively COTS components, with the advantage of allowing the security kernel to be easily deployed on widely used platforms.

1 Introduction

We describe the design of a security kernel called Trusted Timely Computing Base (TTCB). A security kernel is a fail-controlled subsystem trusted to execute a few functions correctly, albeit immersed in an environment subjected to malicious faults. It may be used as an intrusion prevention device, by supporting the mediation/protection of all system interactions, and/or all accesses to system resources. The reference monitor paradigm is such an example [27]. Alternatively, it may be used as an intrusion tolerance device, by considering that interactions are performed in unprotected environments, and are subjected to intrusions. The security kernel intervenes only in crucial phases of execution, in principle to support intrusion tolerance mechanisms, and as such it can be a fairly simple component. Intrusion tolerance is the approach taken in the MAF-

TIA project [35], under which the TTCB is being developed[1]. The TTCB assists the implementation of some of the intrusion-tolerant middleware components, whose architecture and general design principles are described in [44].

The TTCB has innovative features. Firstly, it is a distributed subsystem with its own secure network. A distributed security kernel represents a 'hard-core' component, offering trusted services to a collection of participants, despite the fact that the latter reside in different nodes, and that their normal communication is through an insecure network. In consequence, the collection of participants can achieve some degree of distributed trust, for low-level facts reported to/by the TTCB for/to all (and thus agree on them), without having to explicitly communicate. That is, protocol participants essentially exchange their messages in a world full of threats, some of them may even be malicious and cheat, but there is an oracle that correct participants can trust, and a channel that they can use to get in touch with each other, even for rare moments. Moreover, this oracle also acts as a checkpoint that malicious participants have to synchronize with, and this limits their potential for Byzantine actions (inconsistent value faults).

Secondly, the TTCB is synchronous (or real-time), in the sense of having reliable clocks and being able to execute timely functions, and obviously do it in a distributed way: the control channel provides timely (synchronous) inter-module communication. As such, it is capable, for example, of telling the time, measuring durations, and detecting timing failures.

Thirdly, the TTCB can be implemented using only COTS components (operating system and hardware). In consequence, all the design guidelines and the mechanisms we describe in the paper are reproducible and useable in open settings. As a matter of fact, we are going to make available a prototype of the TTCB for free non-commercial use, for developers of intrusion-tolerant protocols.

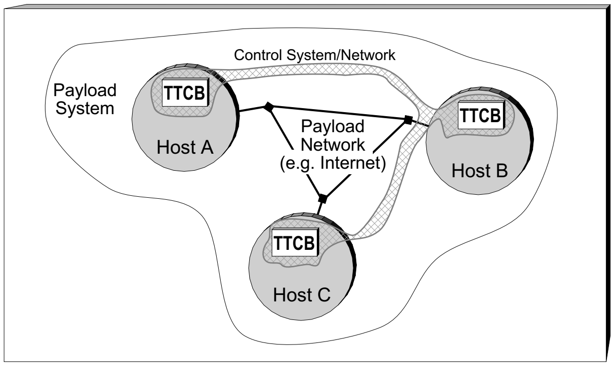

Figure 1: The architecture of a system with a TTCB

We discuss essentially three things in this paper about our distributed real-time security kernel:

(1) The TTCB is a simple component providing a small set of basic secure services. It aims at building a new style of protocols to achieve intrusion

[1] For information, check the MAFTIA home page in www.maftia.org

tolerance, which for the most part execute in insecure, arbitrary failure environments, and resort to the TTCB (used here as an oracle and a facilitator) only in crucial parts of their operation.

(2) The TTCB is a synchronous subsystem supplying time-related functions, such as timing failure detection, in a trusted way. In consequence, it may be an enabler of a new generation of timed secure protocols, until now known to be fragile due to attacks on timing assumptions.

(3) The TTCB follows a design methodology based on a composite fault model along the attack-vulnerability-intrusion trilogy. The methodology establishes our hybrid failure assumptions in a well-founded manner, by typifying faults and assessing coverage. Besides, although the TTCB could be designed using tamperproof hardware, we show here that high enough coverage can still be achieved using exclusively COTS components, with the advantage of allowing the security kernel to be easily deployed on widely used platforms.

In another paper we show how the basic services mentioned above are used to implement efficient Byzantine-resilient protocols, i.e., intrusion-tolerant protocols [15]. In a previous paper, we had shown how to materialize timed protocols in uncertain timeliness environments [43].

The paper is organized as follows. Section 2 introduces the TTCB and Section 3 gives answers to the question: what is the TTCB good for? The latter section mentions the security services of the TTCB, therefore the next two sections describe the design of these services. Section 4 is about the local security services and Section 5 is about the single distributed security service. Section 6 is about the time services. These three sections are about the functional design of the services, i.e., of the protocols and algorithms that make they perform their services. The rest of the paper is about the non-functional design of the TTCB, i.e., of how the design guarantees the TTCB non-functional properties, timeliness and security (fail-silent). Section 7 is about the design methodology. The following section, 8, gives the architecture of the design based on COTS. Section 9 gives the environment assumptions and the mechanisms that enforce these assumptions. Section 10 and 11 are respectively about the design of the network and local TTCB. Section 12 discusses the related work and Section 13 concludes the paper.

2 The TTCB

The TTCB is a secure real-time distributed component that aims to assist the execution of applications. The architecture of a system with a TTCB is suggested in Figure 1. An architecture with a TTCB has a local module in some hosts, called the *local TTCB*. These modules are interconnected by a *control channel*. This set up of local TTCBs interconnected by the control channel is collectively called *the* TTCB. The TTCB is used to assist protocols and applications running between participants in the hosts concerned, on any usual distributed system architecture, encompassing a set of hosts interconnected by a network (e.g., the Internet). We call the latter the *payload system and network*, to differentiate from the TTCB part.

Conceptually, a *local TTCB* should be considered to be a *module* inside a host, with a well defined interface and separated from the OS. In practice, this conceptual separation between the local TTCB and the OS can be achieved in several ways: (1) the local TTCB can be implemented in a separate, tamperproof hardware module —coprocessor, PC board, etc.— and so the separation is physical; (2) the local TTCB can be implemented on the native hardware, with a virtual separation and shielding implemented in software, between the former and the OS processes. The design of the TTCB is discussed later in the paper.

AN1 Broadcast – The AN has an unreliable packet broadcast primitive (the sender also receives)

AN2 Integrity – Nodes can detect if packets were corrupted in the network. Corruptions are converted to omission failures

AN3 Omission degree – No more than *Od* omissions may occur in a given interval of time

AN4 Bounded delay – Any correct packet is received within a maximum delay T_{send} from the send request

AN5 Partition free – The network does not get partitioned

AN6 Broadcast Degree – If a broadcast is received by any local TTCB other than the sender, then it is received by at least *Bd* local TTCBs

AN7 Confidentiality – The content of network traffic cannot be read by unauthorized users

AN8 Authenticity – Nodes can detect if a packet was broadcast by a correct node

Table 1: Abstract Network properties.

The local TTCBs are assumed to be fail-silent (they fail by crashing). The TTCB cannot produce erroneous interactions or results (even on account of attacks). Every local TTCB has a clock and the clocks are synchronized: the clock values of correct processes at any real-time t differ by at most a known constant π, the *precision* of the clock set.

The TTCB control channel has well-defined characteristics, specified in Table 1 as a set of abstract network properties, on which the design of the internal protocols relies. In this way the control channel does not have to rely on a specific network technology: the abstract network can be mapped onto different networks with the assistance of simple adaptation mechanisms.

The TTCB offers two sets of services, listed in Table 2, which any component (protocol, application) in the local host can use. These services are described with detail in [33]. In the table and throughout the paper we use the word *entity* to denominate any software component that calls the TTCB (process, thread, etc.).

3 Intrusion Tolerance with the TTCB

Before we delve into the discussion of these services, a pertinent question at this stage is: *What is the TTCB good for?* This question is best answered after explaining the failure assumptions followed in the MAFTIA architecture.

Security services	
Local authentication	For an entity to authenticate the TTCB and establish a secure channel with it.
Trusted block agreement	Achieves agreement on a small, fixed size, block of data.
Trusted random number generation	Generates trustworthy random numbers.
Time services	
Trusted timely execution	Executes operations securely and within a certain interval of time.
Trusted duration measurement	Measures the duration of the execution of an operation.
Trusted timing failure detection	Checks if an operation is executed during an interval of time.
Trusted absolute timestamping	Provides globally meaningful timestamps.

Table 2: TTCB Services

3.1 Fault Model

A crucial aspect of any fault-tolerant architecture is the fault model upon which the system architecture is conceived, and component interactions are defined.

Controlled failure assumptions specify qualitative and quantitative bounds on component failures, representing very well how common systems work under the presence of accidental faults. However, it is traditionally difficult to model the behavior of an intruder, so we have a problem of coverage that does not recommend this approach unless a solution can be found.

Arbitrary failure assumptions, on the other hand, specify no qualitative or quantitative bounds on the component's possible failures. Arbitrary failure assumptions are costly to handle, in terms of performance and complexity, and thus are not compatible with the user requirements of the vast majority of today's on-line applications.

Hybrid assumptions, combining both kinds of failure assumptions, are followed in our work. Generally, they consist of allocating different assumptions to different subsets or components of the system, and have been used in a number of systems and protocols [32, 34, 45].

With hybrid assumptions some parts of the system would be justifiably assumed to exhibit fail-controlled behavior, whilst the remainder of the system would still be allowed an arbitrary behavior. This is advantageous in modular and distributed system architectures subjected to malicious faults.

However, such an approach is only feasible when the fault model is well-founded, that is, the behavior assumed for every single subset of the system can be modeled and/or enforced with high coverage. As a matter of fact, a system normally fails by its weakest link, and naive assumptions about a component's behavior will be easy prey to hackers.

The implementation of the TTCB discussed in Sections 10 and 11 combines different techniques and methods tackling different classes of faults, in order to achieve the postulated behavior (fail-silent) with high coverage.

3.2 Strategy for Intrusion Tolerance

With the TTCB, we can implement intrusion-tolerance mechanisms, on a hybrid of arbitrary-failure (the payload system) and fail-silent (the TTCB) components.

The TTCB is designed to assist crucial steps of the operation of middleware protocols. We use the word "crucial" to stress the tolerance aspect: unlike classical, prevention-based approaches (e.g., Reference Monitor), the component does not stand in the way of all resources and operations. As a matter of fact, protocols run in an untrusted environment, local participants only trust interactions with the (trusted) security kernel, single components can be intruded, and correct service provision is built on distributed fault tolerance mechanisms, for example through agreement and replication amongst collections of participants in several hosts.

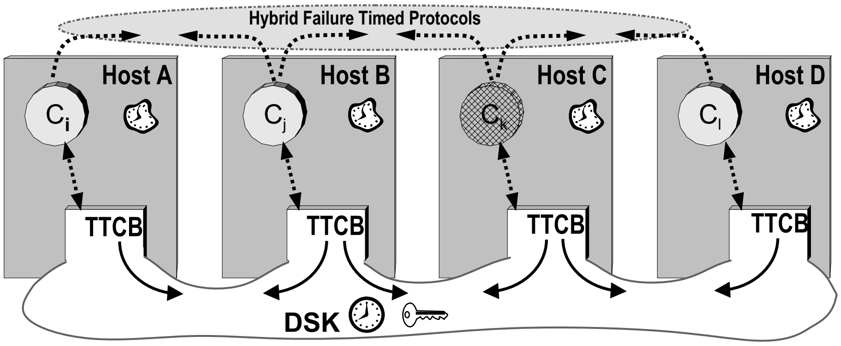

Figure 2: Intrusion Tolerance with a TTCB

Observe Figure 2: software components C_i interact through protocols which run on the payload system (the top arrows). However, they can locally access the TTCB in some steps of their execution (for example, to be informed whether a message just received was or not corrupted). The white colour is used to mean a trusted environment (the TTCB). The key means the environment is cryptographically secure. The grey colours for the payload system mean untrusted. The time-related issues depicted in the figure will be discussed in Section 6.

Trusting the TTCB security kernel means the following: it is not feasible to subvert the TTCB, but it may be possible to interfere in its interaction with entities. In similar terms, whilst we let a local host be compromised, we must make sure that it does not undermine fault-tolerant operation of the protocols amongst distributed components.

The above implies two things: the operation of protocols can be intruded upon and individual components can be corrupted (e.g., C_k); and special care must be taken in order to preserve the validity of the interactions of a correct entity with its local TTCB. The reader is referred to [15], where we give a practical example of the use of the TTCB to implement intrusion-tolerant protocols.

4 TTCB Local Security Services

This section describes the local security-related services of the TTCB (Local Authentication Service and Random Number Generation Service) and the entity–TTCB communication.

4.1 Local Authentication Service

The purpose of the Local Authentication Service is to allow the entity to authenticate and establish a *secure channel* with a local TTCB. The need for this service derives from the fact that, in general, the communication path between the entity and the local TTCB is not trustworthy. For instance, that communication is probably made through the operating system that may be corrupted and behave maliciously. We assume that the entity–local TTCB communication can be subject to passive and active attacks [31]. A call to the TTCB is composed of two messages, a request and a reply, that can be read, modified, reordered, deleted, and replayed.

Every local TTCB has an asymmetric key pair (K_u, K_r) that is used to authenticate it. The entity that calls the Local Authentication Service is assumed to have an authentic copy of the local TTCB public key K_u. These public keys can be distributed, for instance, manually or using a Public Key Infrastructure (PKI). The private key K_r is assumed to be known only by the local TTCB. A secure channel is obtained establishing a shared symmetric key K_{et} between the entity and the local TTCB, that is later used to secure their communication (Section 4.3).

The protocol to establish the shared secret has to be an *authenticated key establishment protocol* with local TTCB authentication. Such a protocol is defined by the following properties [31]:

- *SK1 Implicit Key Authentication.* The entity and the TTCB know that no other entity has the key.
- *SK2 Key Confirmation.* Both the entity and the TTCB know that the other has the key.
- *SK3 Authentication.* The entity has to authenticate the local TTCB.
- *SK4 Trusted Against Known-Key Attacks.* Compromise of past keys does *not* allow either (1) a passive adversary to compromise future keys, or (2) impersonation by an active adversary[2].

A simple protocol with properties SK1 through SK4 can be implemented with two messages, i.e., a single function call. Figure 3 shows the protocol. The proof of the protocol is in Appendix A.

		Action	**Description**
1	P → T	$\langle E_u(K_{et}, X_e)\rangle$	The entity sends the TTCB the new key K_{et} and a challenge X_e, both encrypted with the local TTCB public key K_u
2	T → P	$\langle S_r(X_e)\rangle$	TTCB sends the entity the signature of the challenge obtained with its private key K_r

Figure 3: Local Authentication Service protocol

The protocol requires the entity to generate the key K_{et}. Although we would desire the key to be generated by the local TTCB, there is no key that

[2] A passive adversary "attempts to defeat a cryptographic technique by simply recording data and thereafter analyzing it (e.g., in key establishment, to determine the key). An active attack involves an adversary who modifies or injects messages." [31]

the TTCB can use to protect and give K_{et} to the entity. Therefore K_{et} has to be generated by the entity but in such a way that a malicious OS cannot guess or disclose it. The generation of a random key requires sources of randomness (timing between key hits and interrupts, mouse position, etc.), sources that in mainstream computers are controlled by the OS. This means that when an entity gets allegedly random data from those sources, it may get either data given or known by a potentially malicious OS. Therefore, there is the possibility of a malicious OS being able to guess the random data that will be used by the entity to generate the key, and consequently, the key itself. This problem is hard to handle. A set of criteria can help to mitigate it:

- the entity should use as much as possible sources of random data not controlled by the OS.
- The entity should use as much different sources of random data as possible. Even if an attacker manages to corrupt the OS, it will probably not be able to corrupt its code in many different places and in such a synchronized way, so that it may guess the random number.
- The entity must use a *strong mixing function*, i.e., a function that produces an output whose bits are uncorrelated to the input bits [20]. An example is a hash function such as MD4 or MD5.

Although this may seam limitative, for most applications it is reasonable to take some initial time to obtain a random number. In runtime, this will in general be unacceptable and that is why it is useful to have the Random Number Generation Service, that gives reliable random numbers, provided that a secure channel was previously established. A final comment: the protocol challenge should also be generated randomly using the same set of criteria.

The Local Authentication Service protocol is implemented in the TTCB API as a single call with the following syntax:

`eid, chllg_sign` ← **TTCB_localAuthentication**`(key, protection, challenge)`

The input parameters are the key, the communication protection to be used (see Section 4.3), and the challenge. The input parameters are encrypted with the local TTCB public key. The output parameters are the entity identification –*eid*– used to identify the entity in the subsequent calls, and the signature of the challenge, obtained using the local TTCB private key.

4.2 Random Number Generation Service

The Trusted Random Number Generation service supplies uniformly distributed random numbers. These numbers are basic for building cryptographic primitives such as authentication protocols. If the numbers are not really random, those algorithms can be vulnerable.

The interface of the service has only one function that returns a random number:

`number` ← **TTCB_getRandom**`()`

In a future version of the TTCB, based on a appliance board, we envisage the use of a hardware random number generator, since it can give better random numbers. In the current RT-Linux TTCB, the random numbers are given by Linux `/dev/random` device. This device works with an entropy pool that collects random data from several inputs: device driver noise, timing between key hits, timing between some interrupts, mouse position, timing between disk accesses, etc. When a random number is requested, a hash of the entropy poll is calculated using MD5.

4.3 Entity-TTCB Secure Channel

The communication between an entity and the local TTCB is protected by a secure channel, based on the existence of the shared key K_{et}. It is reasonable to consider that entities can have different security requirements for this communication. Therefore, when the Local Authentication Service is called, the parameter *protection* is used to select one of three *protection modes* below. (The same parameter is also used to select encryption and hashing algorithms, if necessary.)

1. *Authenticity only.* In particular situations or system architectures, an entity may be able to communicate securely with its local TTCB. This may be the case if the entity is the OS itself, or if the local TTCB is called through an I/O port. Therefore, the secure channel has only to guarantee the authenticity of calls to the local TTCB, i.e., that the service is being called by the entity whose *eid* comes in the call.

2. *Authenticity and integrity.* The entity requires only that the communication integrity is guaranteed.

3. *Authenticity, integrity and confidentiality.* The entity requires not only that the communication integrity is guaranteed but also that its content cannot be disclosed.

In the first mode, *authenticity* is guaranteed putting the key K_{et} together with the *eid* in the messages sent from the entity to the local TTCB. This allows the TTCB to authenticate the caller entity. Attacks against authenticity in the second and third modes of protection are handled including the *eid* in the message. Together with the Message Integrity Code (MIC) used to enforce integrity, this authenticates the message.

Ensuring the message *integrity* – second and third modes – is more complex. Below is a list of the possible attacks and how each one is handled:

- *Modification.* Message modification is detected using a Message Integrity Code (MIC). The algorithm used to obtain the MIC is defined using the parameter *protection.* The algorithm can be, for example: “calculate a hash of the message using SHA-1 and encrypt it with 3DES”. If the local TTCB detects that a request message was corrupted, it returns an error. If an entity detects a corrupted reply message, it simply discards it. There is no point in resending messages since modifications have a high probability of having a malicious causer and there is a high probability that this causer is able to corrupt any number of resends. Therefore, these situations have to be solved by the entity.

- *Replay.* Every request message includes a sequence number. The corresponding reply includes the same number. Therefore, both the local TTCB and the entity know if the message they received is the one it should be or a replay. Replay are simply dropped.

- *Reorder.* The reorder of messages is detected and handled as replays.
- *Deletion.* If a message is deleted the entity will not receive the response to a call it made. This case has to be handled by the entity (e.g., using a timeout to detect if its communication with the local TTCB is attacked).

Confidentiality (third mode) is enforced encrypting the message with the key K_{et}.

5 TTCB Distributed Security Services

Distributed services are services that require the cooperation of several local TTCBs for their execution. This section describes the only TTCB distributed security-related service— the Trusted Block Agreement service— but a fundamental one. The remainder distributed services are time-related, such as Timing Failure Detection [11], and will be briefly addressed in Section 6.

5.1 The Trusted Block Agreement Service

The Trusted Block Agreement Service (Agreement Service for short) performs agreement protocols between sets of entities. These protocols, for instance multicast or consensus with a majority decision, are executed in a secure and timely fashion, since the service runs inside the TTCB. The service is not supposed to replace agreement protocols in the payload system: it works with "small" blocks of data (currently 160 bits), and the TTCB has limited resources to execute it.

The Agreement Service is formally defined in terms of the three functions *TTCB_propose*, *TTCB_decide* and *decision*. A entity is said to *propose a value* when it calls *TTCB_propose*. A entity *decides a result* when it calls *TTCB_decide* and receives back a result. The function *decision* defines how the result is calculated in terms of the inputs of the service. The *result* is composed by a value and some additional information that will be described below. Formally, the Agreement Service is defined by the following properties:

- *Termination.* Every correct entity eventually decides a result.
- *Integrity.* Every correct entity decides at most one result.
- *Agreement.* If a correct entity decides *result*, then all correct entities eventually decide *result*.
- *Validity.* If a correct entity decides *result* then *result* is obtained applying the function *decision* to the values proposed.
- *Timeliness.* Given an instant *tstart* and a known constant $T_{agreement}$, a process can decide by $tstart + T_{agreement}$.

It is important to note that Timeliness is a property valid only at the TTCB interface. An entity can only decide with the timeliness the payload system permits.

The interface of the Agreement Service has two functions, *TTCB_propose* and *TTCB_decide*, that correspond to those in the definition. An entity calls *TTCB_propose* to propose its value and *TTCB_decide* to try to decide a result (*TTCB_decide* is non-blocking and returns an error if the agreement did not terminate).

```
out ← TTCB_propose(eid, elist, tstart, decision, value)
result ← TTCB_decide(eid, tag)
```

We use the expression *an agreement* to denominate an execution of the Agreement Service. An agreement is uniquely identified by three parameters: *elist* (the list of entities involved in the agreement), *tstart* (a timestamp), and *decision* (a constant identifying the decision function). The service terminates at most $T_{agreement}$ after it "starts", i.e., after either: (1) the last entity in *elist* proposed; or (2) after *tstart*. That shows the meaning of *tstart*: it is the instant at which an agreement "starts" despite the number of entities in *elist* that proposed. If the TTCB receives a propose after *tstart* it returns an error.

Let us see the other parameters, starting with *TTCB_propose*. *eid* is the unique identification of an entity before the TTCB, obtained using the Local Authentication Service. *value* is the block the entity proposes. *out* is a structure with two fields: *error*, an error code; and *tag*, an unique identifier of the agreement before a local TTCB. An entity calls *TTCB_decide* with the *tag* that identifies the agreement that it wants to decide. *result* is a record with four fields: (1) *error*, an error code; (2) *value*, the value decided; (3) *proposed-ok*, a mask with one bit per entity in *elist*, where each bit indicates if the corresponding entity proposed the value that was decided or not; (4) *proposed-any*, a similar mask that indicates which entities proposed any value.

Currently we defined only a few *decision* functions:

- *TTCB_TBA_RMULTICAST*. This function, on one hand, is a reliable multicast since the entities decide the value proposed by the first entity in *elist* (the others may not propose a value at all). On the other hand, the function also compares the value decided with the values proposed by the other entities (if any) and returns the masks *proposed-ok* and *proposed-any*.

- *TTCB_TBA_MAJORITY*. This function returns the value proposed more times and the two masks.

- *TTCB_TBA_TEQUALITY*. This function compares the values proposed. The entities that proposed the value more proposed decide the result with the masks. The others receive an error.

5.2 Agreement Service Protocol

This section describes an Agreement Service protocol. The protocol is time-triggered: *TTCB_propose* is called asynchronously, and gives the TTCB data that is stored in tables; then, periodically that data is broadcast to all local TTCBs and, also periodically, data is read from the network and processed.

The protocol uses two tables. The *dataTable* stores all agreements data. Each record has the state of one agreement with the format: *(tag, elist, tstart, decision, vtable)*. All fields have the usual meaning except *vtable*, which is a table with the values proposed (one per entity in *elist*). *sendTable* stores data

```
For each local TTCB
propose routine
1  when entity calls TTCB_propose(eid, elist, tstart, decision, value) do
2      if (entity already proposed) or ($eid \notin elist$) or ($clock() > tstart$) then return error;
3      insert (elist, tstart, decision, eid, value) in sendTable;
4      get $R \in dataTable : R.elist = elist \wedge R.tstart = tstart \wedge R.decision = decision$;
5      if ($R = \perp$) then $R :=$ (get_tag(), elist, tstart, decision, $\perp$); insert R in dataTable;
6      return R.tag;
broadcast routine
7  when $clock() = round_s \times T_s$ do
8      repeat $Od + 1$ times do broadcast(sendTable);
9      $sendTable := \perp$; $round_s := round_s + 1$;
receive routine
10 when $clock() = round_r \times T_r$ do
11     while ($read(M) \neq error$) do
12         foreach (elist, tstart, decision, eid, value) $\in M.sendTable$ do
13             get $R \in dataTable : R.elist = elist \wedge R.tstart = tstart \wedge R.decision = decision$;
14             if ($R = \perp$) then $R :=$ (get_tag(), elist, tstart, decision, $\perp$); insert R in dataTable;
15             insert value in R.vtable;
16     $round_r := round_r + 1$;
decide routine
17 when entity calls TTCB_decide(tag) do
18     get $R \in dataTable : R.tag = tag$;
19     if ($R \neq \perp$) and [($clock() > R.tstart + T_{agreement}$) or (all entities proposed a value)] then
20         return (calculate result using function R.decision and values in R.vtable);
21     else return error;
```

Figure 4: Agreement Service internal protocol. Instance at a local TTCB.

to be broadcast to all local TTCBs. Every record is a proposal with the format: *(elist, tstart, decision, eid, value)*. The agreement is identified by *(elist, tstart, decision)*. *eid* identifies the entity that proposed and *value* is the value proposed.

Figure 4 shows the protocol. It is based on TTCB assumptions that we summarize here for clearness: (1) the local TTCBs have clocks synchronized to π; (2) the protocol code is executed in real-time (therefore there is a worst case execution time for every section of code); (3) every local TTCB communicates with the others exclusively by broadcasting a message with a constant period; (4) the network is described by the Abstract Network model (Table 1).

The protocol has four routines. The *propose routine* is executed when an entity calls the TTCB function *TTCB_propose* (Lines 1-6). The routine begins doing some tests: if the entity already proposed a value for this agreement; if the entity that calls the service is in *elist*; if *tstart* already expired (Line 2). Other tests, are also made but are not represented since they are not related to the algorithm functionality. If the propose is accepted, its data is inserted in sendTable and dataTable, and the *tag* is returned (Lines 3-6). The *broadcast routine* broadcasts data to all local TTCBs every T_s (the period) either if there is data in sendTable or not (Lines 7-9). Every message is broadcasted $Od+1$ times in order to tolerate omissions in the network (Od is the omission degree). After the broadcast, sendTable is cleaned. The *receive routine* reads and processes messages every T_r (Lines 10-16). Since each message is broadcasted $Od + 1$ times, copies of the same message have to be discarded by the function *read* (Line 11). For each message received, the data in each record of sendTable is inserted in dataTable (Lines 12-15). The *decide routine* is executed when an entity calls the function *TTCB_decide*. The routine searches dataTable for the agreement identified by the tag and returns an error if it does not exists. If the instant $tstart + T_{agreement}$ passed or the local TTCB has the values proposed by all entities in *elist*, the result is obtained and returned.

In Appendix A we prove that the protocol implements the Agreement Service and that $T_{agreement}$ can be given by:

$$T_{agreement} = T_s + WCET_{send} + T_{send} + T_r + WCET_{receive} + \pi \qquad (1)$$

The constants in the formula have the following meaning: T_s and T_r are respectively the send and receive periods; $WCET_{send}$ and $WCET_{receive}$ are respectively the send and receive routines worst execution times; T_{send} is the maximum communication delay; π is the precision of the clock synchronization algorithm.

5.3 Agreement Service Protocol With Local TTCB Crashes

In the Agreement Service protocol in Figure 4, if a local TTCB crashes during the broadcast, some local TTCBs may receive the message while others may not. Such an inconsistency can lead to different local TTCBs giving different results to one or more agreements. Therefore, informally, when the sender crashes, the broadcast must either deliver the message to all recipients or to none. Such a broadcast is usually called a Reliable Broadcast and this section describes such a protocol. If we replace lines 8 and 11 in Figure 4 by this protocol, the Agreement Service protocol becomes tolerant to local TTCB crashes. The second condition in Line 19 has also to be substituted by: "all entities *in non-crashed local TTCBs* proposed a value". The complete protocol is shown in Appendix B.

This section presents a time-triggered Timely Reliable Broadcast that tolerates crashes, assumes channel omissions (Abstract Network property AN3), and is lightweight, in the sense that it does not retransmit messages. A timely reliable broadcast is formally defined in terms of two primitives *R-broadcast (M)* and *R-deliver (M)*, where M is the message, that verify the following properties (based on [22]):

- *Validity.* If a correct local TTCB R-broadcasts M then it eventually R-delivers M.
- *Agreement.* If a correct local TTCB R-delivers message M then all correct local TTCBs eventually R-deliver M.
- *Integrity.* For any message M, a correct local TTCB R-delivers M at most once and only if M was R-broadcast by *sender (M)*.
- *Timeliness.* There is a known constant $T_{broadcast}$ such that, if a message is R-broadcast at instant t, then no correct local TTCB R-delivers M after $t + T_{broadcast}$.

The protocol is shown in Figure 5. The *broadcast routine* is similar to the Agreement Service protocol. Every T_s a message M is broadcasted $Od+1$ times to tolerate omissions in the channel. The message is broadcasted even if there is no data to be sent. This is important for the protocol to work properly and for the detection of local TTCB crashes (a local TTCB is known to be crashed if a message is not received by its deadline [11]). The message has an header with the sender identifier, a sequence number and the table *higherseqVector*. This table has an entry for every local TTCB that contains, for every other local TTCB,

For each local TTCB

```
broadcast routine
1   when clock() = $round_s \times T_s$ do
2         sender := my_id(); seq := $round_s$;
3         M := (sender, seq, higherseqVector, data);
4         repeat $Od + 1$ times do broadcast(M);
5         $round_s$ := $round_s$ + 1;
receive routine
6   when clock() = $round_r \times T_r$ do
7         while (read(M) $\neq$ error) do
8               foreach M-ndlv in notDelivered do
9                     if [(M-ndlv.sender = M.sender) and (M-ndlv.number < M.number)] or
                              (M-ndlv.number < M.higherseqVector[M-ndlv.sender]) then
10                          R-deliver(M-ndlv.data); remove M-ndlv from notDelivered;
11              if (higherseqVector[M.sender] > M.number) then R-deliver(M.data);
12                    else put M in notDelivered; higherseqVector[M.sender] := M.number;
13        $round_r$ := $round_r$ + 1;
```

Figure 5: Timely reliable broadcast protocol.

the highest sequence number of a message received from that local TTCB. The *receive routine* starts by reading a message M (Lines 6-13). Copies of messages already received are discarded by the function *read*. For every message received the routine does two things: (1) tests if previously received but not R-delivered messages (stored in *notDelivered*) can be R-delivered (Lines 8-10); (2) tests if M can be R-delivered (Lines 11-12).

Considering AN6, the protocol tolerates Bd local TTCB crashes in a reference interval of time. A message can be R-delivered by a local TTCB when it knows that all other non-crashed local TTCBs will also R-deliver it (Agreement property). A local TTCB can R-deliver a message $M(s, n)$ when it receives (a) $M(s, n+1)$ or (b) $M(s', n')$ with *higherseqVector[s]=n+1* (s is the sender and n the message number). The intuition behind this is: if s crashes during the broadcast of $M(s, n+1)$ but at least one local TTCB receives the message, then at least Bd local TTCBs receive it (AN6) and at most other $Bd-1$ can crash (the protocol tolerates Bd crashes). Therefore, at least one correct local TTCB receives $M(s, n+1)$ and broadcasts $M(s\,', n\,')$ with *higherseqVector[s]=n+1* to the other non-crashed local TTCBs. Since messages are broadcast $Od+1$ times, all non-crashed local TTCBs either receive $M(s, n+1)$ or $M(s\,', n\,')$ and R-deliver $M(s, n)$. In the protocol, Line 9 tests this condition. However, it considers that any $M(s, n_+)$ with $n_+ > n$ causes $M(s, n)$ to be R-delivered, since the Abstract Network does not guarantee the order of the reception of messages. The same is true for $M(s\,', n\,')$ with *higherseqVector[s]* $> n$. Line 11 checks if the message received, $M(s\,'', n\,'')$, can be R-delivered immediately. This is the case if a message from the same sender but with a higher number was received previously, i.e., if *higherseqVector[s'']* $> n''$.

In Appendix A we prove these results and also that the protocol R-delivers a message M within $T_{broadcast}$ of *R-broadcast(M)* (the meaning of the constants is the same as before):

$$T_{broadcast} = 2 \times (WCET_{send} + T_{send} + T_r + WCET_{receive} + T_s) + \pi \qquad (2)$$

The Agreement Service termination instant is related to $T_{broadcast}$:

$$T_{agreement} = T_s + T_{broadcast} \tag{3}$$

The proof of this result is in the Appendix A.

6 TTCB Time Services

This section addresses the time-related services of the TTCB. The latter are discussed with detail in [33]. Due to lack of space, we briefly underline the potential of these services.

In order to understand the assumptions on timeliness of our system, let us go back to Figure 2: the clock inside the TTCB area is meant to suggest it is a fully synchronous (or hard real-time) component. On the other hand, the warped clock in the payload area suggests that it has uncertain timeliness, or partial synchronism. It can even be asynchronous.

This is a realistic assumption for Internet-based operation, but constructing timed protocols in environments of uncertain timeliness is known to be a hard task. We have already shown [43, 12] how to use the services provided by a timely computing base to solve such problems. The solution lies on the adequate interaction of an essentially asynchronous system with a synchronous subsystem (interface), so as to determine interesting facts about time (services).

Constructing secure timed protocols is an even harder task, due to the risk of attacks on the timing assumptions. For that reason, most known secure broadcast or Byzantine agreement protocols are of the asynchronous class. In the TTCB work, we extend the resilience of the aforementioned time services, to ensure that they can be trusted despite the presence of malicious faults. In summary, the TTCB time services are:

- *Trusted timely execution.* This service executes operations securely and within a certain interval of time. It can be useful to run operations that are critical, in terms of security and/or time, in a secure and timely way.

- *Trusted duration measurement.* This service measures the duration of the execution of an operation, or a sequence of similar operations. It can be useful for instance to help in the detection of attacks against performance or QoS, such as denial-of-service attacks.

- *Trusted timing failure detection.* This service detects if a timed operation fails, i.e., if an operation – local or remote – fails to be executed within an interval of time. If so, a function can be timely executed to handle the failure exception. This service can be extremely important to detect and handle attacks that try to break timeliness assumptions of protocols.

- *Trusted absolute timestamping.* This service provides globally meaningful timestamps. The service is important to implement security systems that rely on reliable timestamps given by synchronized clocks, e.g., Kerberos. Intrusion-tolerant protocols based on the TTCB security-related services also need these timestamps [15].

In consequence, we end-up with an environment where the TTCB, with the security-related and time-related services combined, can support the construction of secure *and* timed protocols.

7 TTCB Design Methodology

This section describes our design methodology. As any design, it has functional and non-functional aspects. The functional aspects are mainly concerned with the algorithms and protocols that make the system perform its service (discussed in Sections 4 and 5), and the necessary adaptations to the environment to ensure it has the run-time functionality assumed by those protocols, for instance, the abstract network. We briefly discuss the latter, and concentrate on the *non-functional* aspects.

Design Principles of a Security Kernel

The construction of a security kernel like the TTCB is based on three design principles:

- *Interposition:* it must by construction be interposed between vital resources and any attempt to interact with them.
- *Shielding:* the TTCB construction is such that it itself is protected (1) from faults affecting timeliness and (2) from faults affecting security, i.e., timing faults, attacks and intrusions.
- *Validation:* the TTCB functionality is simple enough to be based on verifiable mechanisms with relation both to timeliness and security.

The "vital resources" mentioned in *Interposition* are those resources needed for the TTCB itself to behave as specified, i.e., timely and securely. *Shielding* substantiates the principle that attacks against the inside of the TTCB are prevented from being successful. The design principle of *Validation* implies that the TTCB design and implementation has, in some sense, to be "simple", so that it can be verified and justifiably claimed correct. For example, the number of lines of code and the amount of data that it handles have to be limited; the structure of internal procedures and protocol should be simple.

Composite fault model with hybrid failure assumptions

The organization of assumptions in terms of a composite fault model [44] underpins our design philosophy. In MAFTIA, we say that the impairments that may occur to a system, security-wise, have to do with a wealth of causes, which range from internal faults (i.e., vulnerabilities), to external, interaction faults (i.e., attacks) which activate those vulnerabilities, producing faults (i.e., intrusions) that can directly lead to component failure.

The composite fault model is shown in Figure 6. The figure also shows where to apply different techniques to prevent the system from failing. Because we differentiated the several fault classes, we can apply these techniques selectively, and in a structured way.

Note for example, that an intrusion cannot occur unless there is a vulnerability to be activated by a corresponding attack (it makes no sense to prevent an attack for which there is no vulnerability, or vice-versa).

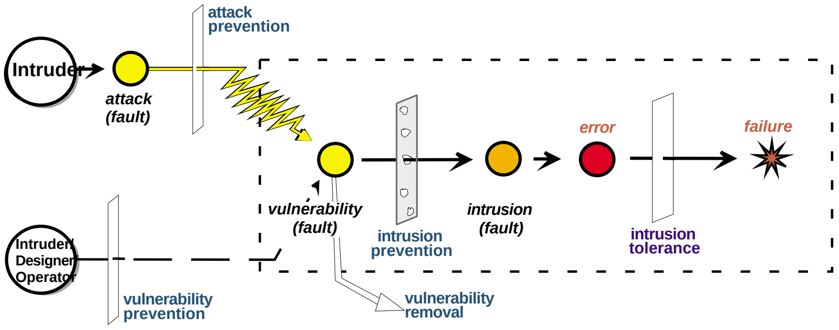

Figure 6: The Composite Fault Model of MAFTIA

A *composite fault model with hybrid failure assumptions* is one where the presence and severity of vulnerabilities, attacks and intrusions varies from component to component.

Consider a component or sub-system like the TTCB, for which a given controlled failure assumption was made. How can we achieve coverage of such an assumption, given the unpredictability of attacks and the elusiveness of vulnerabilities?

The first-line techniques are vulnerability prevention (e.g., using correct coding practices), and then attack prevention (e.g., physically isolating an access point) and vulnerability removal (e.g., patching the OS and removing absolute privileges from the root account).

All these techniques contribute to intrusion prevention. However, after this step there may still be attack-vulnerability combinations to fear from (illustrated in the figure, by the holes in the intrusion prevention barrier). The design must then be complemented with the necessary intrusion tolerance measures, for example, using intrusion detection and recovery or masking, until we justifiably achieve confidence that the component behaves as assumed, failing in the assumed controlled manner, i.e., the component is trustworthy. The measure of its trustworthiness is the coverage of the controlled failure assumption.

7.1 The Methodology

The design of the TTCB with regard to the non-functional properties follows the principles underlined above. The methodology has five steps. It makes sense to perform several iterations until the final result.

1. Define the desired system (TTCB) failure modes
2. Define the architecture and the environment assumptions
3. Design the adaptation mechanisms that enforce the environment assumptions
4. Design the mechanisms and protocols that enforce the system failure modes

5. Assess the system design

6. Iterate until successful

Step one is the definition of the TTCB failure modes. Recall that we consider the local TTCBs to be fail-silent, and consider the inter-TTCB communication system to also be fail-silent. The interposition and shielding principles provide a clear framework, to orient the design in order to achieve these objectives.

Step two starts with the definition of the architecture of the system, i.e., its topology, its main components and their interaction. The architecture itself can prevent some attacks against specific components. For example, the control network being physically inaccessible to hackers. The environment on which the TTCB runs is also defined. The *environment* is what is external to the system being designed but that interacts with it: host hardware and OS, networks, attackers, etc.

Further to typifying the technologies used (the *real environment*), we may create an *abstract model* of parts of the environment, on top of which the system is implemented. This allows us to achieve some independence from the real environment (enhancing portability), and/or to improve it some (simplifying higher-level mechanisms). We do that for the control channel, whose abstract network properties we defined in Table 1, and the third step consists of building the necessary adaptation mechanisms to meet abstraction with reality.

Step four deals with constructing the mechanisms and protocols which enforce the fail-silent behavior of the TTCB, on the assumed environment and architecture. The design methodology may recursively be applied to the internal components of the TTCB as part of this step.

Step five consists in assessing the system design, or in this case, the TTCB subsystem. On the one hand, determining whether the coverage of the design assumptions is acceptably high. On the other hand, determining whether given the assumptions, the algorithmics and their implementation provide the specified services. The verification of the TTCB is on-going work in the context of the MAFTIA project, and will be the subject of future reports.

The following sections apply this methodology to the *design of the TTCB*, as a component with controlled failure assumptions. The generic system architecture and the fault model of the TTCB were introduced in Section 2, and the TTCB architecture is detailed in Section 8. Section 9 presents the environment assumptions and discusses some mechanisms to enforce them. Sections 10 and 11 discusses the design of the control channel and of the local TTCB in view of securing the assumed behavior. For completeness, recall that Sections 4 and 5 have described the functional design of the TTCB.

8 Architecture

The overall architecture is quite simple: local TTCBs interconnected by a (private) control network.

For very high coverage, local TTCBs would normally be built on dedicated tamperproof hardware modules with a dedicated network attachment, such that the modules can be plugged with the proper physical isolation into the host hardware. However, in this paper we show a design based on the very COTS hardware and software which runs the payload system (e.g. PCs with Linux), with isolation implemented in software. This provides the opportunity to show the effectiveness of our structured design methodology. The local architecture of the software-based solution consists of a small secure real-time kernel running on the bare machine hardware, on top of which the regular operating system runs (and all the rest of the host software). The local TTCB is basically a privileged and highly protected set of real-time tasks of that kernel.

For the kernel, we use a general purpose real-time executive, RT-Linux, based on a modified Linux kernel [9, 46]. This approach has the advantage of running Linux and Linux applications on top of it, but it controls all hardware resources in order to safeguard the timeliness of real-time tasks, which are used to support the TTCB. As a disadvantage, it can be as insecure as Linux. The TTCB is built on the RT-Linux kernel, and in consequence, our design concentrates mainly on securing the abovementioned design principles that make-up a trustworthy implementation of a security kernel— interposition and shielding— and discussing the validation of that work. Note that the coverage expected from this configuration cannot be worse than hardened versions of known commercial operating systems, since it only addresses the inner kernel and not the operating system as a whole. It may thus constitute a very attractive implementation for open use, for its cost/simplicity/resilience trade-off.

The control channel can also assume several forms exhibiting different levels of timeliness and resilience. On a timeliness side, it should be observed that the bandwidth required of the control channel is bound to be much smaller than that of the payload channel. To pursue the COTS strategy, our architecture is based on fast Ethernet, for campus-wide systems: we provide each host having a TTCB with an extra LAN adapter. In more demanding scenarios w.r.t. scale, one may resort to alternative networks (e.g., VPNs over ISDN or ADSL connections, GSM or UMTS Short Message Service).

9 Environment Assumptions

This section describes the environment assumptions, and the adaptation mechanisms needed to enforce them. The environment has standard PCs with RT-Linux (the hosts), access to the Internet through regular Ethernet (payload network), and a dedicated Fast-Ethernet (control channel). Hosts have two network devices: one for the payload network, another for the control channel. The environment assumptions are shown in Table 3.

Assumptions A1 and A2 impose limits on what the attacker can do inside a host. Assumption A3 states essentially that the host will never become unstable due to accidental execution faults. Assumptions A4 and A5 impose limits on what the attacker can do to the control channel. Assumptions A6 and A7 define the behavior of the control channel vis-a-vis accidental faults: a controlled omission degree, and a partition-free network environment, respectively.

The interposition principle is essentially achieved by the RT-Linux architecture, whereby the RT-Linux-TTCB kernel compound takes control of all vital resources, intercepting all accesses and interactions (including all interrupts and interrupt handling instructions, such as disable/enable interrupts) coming from the rest of the host, and managing all packets coming from the networks.

Assumptions A1, A2, A4, and A5, aim at justifying the shielding principle. Assumption A3 is directly concerned with stating that the interposition of

A1 The host protection mechanisms cannot be reconfigured by any attacker.

A2 The host kernel memory cannot be read or written by any attacker.

A3 The host is fail-silent in the presence of accidental faults.

A4 The control channel access point cannot be read or written by any attacker.

A5 The data on the control channel cannot be read or written by any attacker.

A6 Given a known interval of time, the control channel does not corrupt more than k packets.

A7 There are no partitions in the control channel.

Table 3: Environment assumptions.

the RT-Linux-TTCB compound between the vital hardware and the other processes, remains valid in the occurrence of accidental faults. Indirectly, it is also concerned with the shielding principle, by stating that in the same situation, it will not become at the mercy of an attacker.

RT-Linux and protection

We start describing RT-Linux and discussing related protection issues. RT-Linux [9, 46] is an engineering of Linux, which was modified in order that a real-time executive takes control of the hardware, in order to enforce real-time behavior of some real-time (RT) tasks. RT tasks were defined as special Linux loadable kernel modules (LKMs), so they run inside the kernel. The scheduler was changed to handle these tasks in a preemptive way and to be configurable to different scheduling disciplines. Linux runs as the lowest priority task and its interruption scheme was changed to be intercepted by RT-Linux. Real-time FIFOs are the basic mechanism for communication between and with RT tasks.

From the point of view of protection, RT-Linux is very similar to Linux. One of the main vulnerabilities is the existence of the superuser privileges. The superuser controls all system resources: it can read, modify and delete any file, any position of memory, etc. Most attacks against Unix/Linux machines at some step try —and often manage — to obtain superuser privileges. This is often achievable, e.g., attacking programs with setuid [8], using race conditions [10] or buffer overflows [17].

Recently several Linux extensions and packages appeared that try to handle this problem, limiting the power of the superuser. We use the *Linux capabilities* since they are already part of the kernel [1].

Linux capabilities are extensions of the *Posix capabilities*, which are privileges or access control lists (ACLs) associated with processes, allowing to control how they can manipulate objects, i.e., other processes, files, directories, unnamed pipes, memory, and the system clock.

When the system reboots, process *init* has the full set of capabilities, and handles the alocation of capabilities to all other processes. This is done through a *capability bounding set*, containing the capabilities that can be held by processes in the system. After a capability is removed from that set, it cannot be used by any process in the system until the next reboot, not even by a process with superuser privileges. The capability bounding set can thus be used automatically, during reboot, to permanently limit the privileges of processes until the next reboot. This mechanisms is very limited since it allows only a resource to be enabled or disabled for *all* processes/users. However, it can still be useful for our purposes.

Enforcing environment assumptions

Assumptions A1 and A2 impose the only limits on what the attacker can do inside a host. It can do everything else. We assume that it can access the host, run software there, and become root or run processes with superuser privileges[3].

The protection mechanisms mentioned in A1 are basically a set of commands in a script that remove a set of Linux capabilities from the capability bounding set. This script is executed when the host is rebooted. Therefore, assumption A1 is secured preventing hackers from rebooting the system. This can be done either protecting the access to the host or using a reboot password.

Assumption A2 protects the working space of both the RT-Linux kernel and the RT task supporting the TTCB. If the attacker manages to modify the kernel memory, he has a dramatic potential for damage, which ranges from modifying kernel or TTCB code or state, to arbitrarily controlling any of the system components, since code in the kernel memory can execute privileged CPU instructions.

Assumption A2 is enforced by removing two vulnerabilities. The removal of these vulnerabilities reduces the power of the superuser and consequently, the power of the attacker:

[3] This discussion concerns the environment during runtime. We have also to prevent the kernel and the local TTCB binaries from being corrupted by an attacker. The files can either be ROM-based, or their integrity can be checked whenever the system reboots with an application such as Tripwire.

Loadable kernel modules:

- Loadable kernel modules (LKMs) are the standard way of inserting code in the kernel in runtime. Their insertion and removal is restricted to the superuser but, since we consider that the attacker can become superuser, it is a vulnerability. This vulnerability is removed taking the capability `CAP_SYS_MODULE` off the capability bounding set. The local TTCB has to insert one LKM and several real-time tasks (that are also LKMs) in the kernel. This has to be done during reboot, before the capability is removed from the bounding set.

`/dev/mem` and `/dev/kmem` devices:

- The devices `/dev/mem` and `/dev/kmem` allow an attacker with superuser privileges to change code and data in the kernel. This can be used to change the kernel and local TTCB code and state. This is even more serious since the file `System.map` maps kernel symbols to physical addresses. An example of how this can be used to corrupt the kernel is a vulnerability that was found on the implementation of the capability bounding set itself. The physical address of this variable has the symbol `cap_bset`. A simple grep of `System.map` allows one to get the physical address of the variable and a simple write in the memory allows the modification of the bounding set value. These devices can even be used to insert code in the kernel [14]. This vulnerability is removed disabling access to the two devices. This is done by removing `CAP_SYS_RAWIO` from the capability bounding set.

Assumption A3 says that hypothetical accidental faults do not affect the normal behavior of the host except by crashing it. The stability of Linux is such that we consider it a given. The assumption makes sense since kernel instability on account of these faults (e.g., memory bit corruptions, processor bugs, user process misbehavior) has a very low probability of occurring.

Assumptions A4 through A7 refer to the control channel. Assumption A4 stipulates that an attacker cannot access the control network adapter from inside the host, and in consequence, he can neither send to, nor read and/or intercept packets from, the control network. This is secured by ensuring the interposition principle for the relevant LAN controller: it must depend solely on the TTCB RT task. However, Linux capabilities cannot help here. Therefore, we define two algorithms, respectively for write and read. The algorithms are implemented in the control channel network device driver:

Write algorithm:

- When a local TTCB sends a packet to the network device driver, an unforgeable tag is appended to the packet (a random number); the tag is stored in a table (in the kernel memory).
- When the device driver gets a packet to be sent to the network it looks at the tag; if the tag is stored in the table it sends the packet and removes the tag from the table; if it is not, it discards the packet.

Read algorithm:

- Before reading a packet the local TTCB stores an unforgeable tag in a table (in the kernel memory); when it calls the device driver to read a packet it gives the device driver the tag;
- When the device driver gets a request to read a packet it checks the tag; if the tag is the one previously stored the driver returns a packet and removes the tag from the table; otherwise it returns an error.

Assumption A5, on the other hand, is secured by ensuring that an attacker cannot have physical access to the control network medium devices (cables, switches, etc.). The assumption makes sense if we consider that it is a short-range, inside-premises closed network, connecting a set of servers inside a single institution, with no other connection. We are assuming that the attacker comes from the Internet, through the payload network, without physical access to the servers or control network hardware. Long-range solutions also use technologies such as ISDN VPN or GSM/UMTS, that are hard for the common Internet intruder to tamper with in conjunction with an attack through the payload network. Note however that assumption A5 can still be enforced for a more powerful hacker who can eavesdrop on the control channel, by using cryptographic schemes in the inter-TTCB communication.

In the just assumed absence of active attacks on the control channel, assumptions A6 and A7 establish limits to the events that may affect the timeliness of communication on the former, so that known bounds can be derived on message delivery delays, and failure detection can be accurately performed. Networks can be tested in order to find out the maximum number of packets they may corrupt in an interval of time, the omission degree. Likewise, short-range LANs have negligible partitioning, which can be further improved by using redundant channels, a must to enforce A7 in wider-area networks.

The TTCB can be considered as having one component, the control channel, plus several components of the same type, local TTCBs (Figure 1). In the following sections, we discuss the design of the Abstract Network modeling the control channel, and of the local TTCB.

10 Abstract Network Design

The design of the TTCB control channel must secure the Abstract Network properties listed earlier in Table 1. Let us discuss the necessary adaptation mechanisms to secure some of these properties.

Property AN1 is available in the Ethernet and can be simulated with IP multicast or with several sends in other networks. Property AN2 is imposed by most networks, through the *cyclic redundancy check* (CRC), if no attacks on the network are considered (assumption A5). In case of attacks, message integrity checks (MICs) could be used instead. Property AN3 is guaranteed by the environment assumption A6. A typical value for Od during a protocol execution in short-range LANs is $Od = 1$.

For property AN4 to be guaranteed in a dedicated switched Fast-Ethernet, packet collisions have to be avoided, since they would cause unpredictable delays. This requires: (1) only one host can be connected to each switch port (hubs cannot be used); and (2) the traffic load has to be controlled. This traffic control has to avoid a set of bounds from being exceeded: the switch buffering and switching capacities; the buffering capacity of the network boards used by the local TTCBs; and the network bandwidth. This requirement is achieved in the TTCB since it executes an admission control mechanism whenever the execution of a service is requested, in order to secure timeliness.

Property AN5 is guaranteed by the assumption A7. Property AN6 depends on several factors having to do with the transmission technology, and medium topology. The value of Bd has to be chosen taking these factors in account. In the network we are considering, a switched Fast-Ethernet, the broadcast degree can easily exceed half of the nodes. Properties AN7 and AN8 are guaranteed by assumptions A4 and A5 and could be enhanced using common cryptographic schemes.

11 Local TTCB Design

This section discusses a few issues related with the implementation of local (i.e., non-distributed) services in the RT-Linux TTCB. The local TTCB executive or runtime support environment has real-time behavior: it is able to schedule and timely execute real-time tasks, as long as some pre-defined schedulability conditions are respected. Assumptions A1 and A2 provide a notion of protection. Assumption A3 implies that accidental faults may only crash the local TTCB.

We recall that the software architecture of the local TTCB encompasses security and time services (Table 2). The TTCB API is defined as a set of functions. The functions are defined in a library and communicate with the local TTCB using RT-Linux FIFOs. Each call is transformed in two messages: a *request* and an *reply.* The mechanism is similar to remote procedure calls.

The local TTCB is implemented by a *kernel module* (LKM) and by a number of *real-time tasks* (RT tasks). The LKM handles calls from the entities (i.e., from the library). The LKM is not real-time since it is part of the interface of the TTCB. All operations with timeliness constrains are executed by RT tasks. A local TTCB has always at least two RT tasks that handle communication: one to send messages to the other local TTCBs and another to receive and process incoming messages. Some services use additional RT tasks. This is the case for the Timely Execution service whose purpose is precisely to execute tasks during a user defined intervals of time. The Timing Failure Detection service may also execute a RT task if a timing failure occurs [43] (see Table 2). The random numbers returned by the TTCB cannot be guessed by an attacker due to the environment assumption A2.

12 Related Work

The TTCB is a distributed security kernel [3] which is radically different from the classic Trusted Computing Base (TCB) [42]. The objective of the Trusted Computing Base is to provide *intrusion prevention* for all critical software in the host; the TTCB, on the contrary, is supposed to be the only secure component and to support the implementation of distributed fault- and intrusion-tolerant protocols. Some secure communication systems assume that the host has a TCB that includes the system software and the OS [37, 4]. Other systems are based on Byzantine protocols and are intrusion-tolerant [13, 26, 36, 2, 19]. In another paper we show how the TTCB can be used to implement efficient intrusion-tolerant protocols [15].

The idea of using a secure device to assist the implementation of secure applications is not new. The Trusted Computing Platform Alliance (TCPA) is defining a secure *subsystem* that provides some local security services to the applications [40]. Project Dyad explored the use of the Citadel secure coprocessor to implement a number of secure distributed applications [41]. Several papers describe the use of SmartCards with the same generic purpose [25, 38, 39]. Although this paper describes the implementation of the (local) TTCB in software, it could also be implemented inside devices like secure coprocessors or SmartCards. Moreover, the TTCB is a distributed component and therefore it can support and assist distributed applications in a more effective way [15]. The TTCB is also real-time, so it can assist the execution of applications with time requirements.

The implementation of the TTCB based on removing vulnerabilities from the operating system, builds on several taxonomies of vulnerabilities [28, 5]. Several Linux extensions and packages appeared recently in order to handle the problem of the Unix superuser "omnipotence". Several of these packages are based on the now withdrawn draft standard IEEE 1003.1e (Posix.1e) [24] and provide mechanisms to implement the *least privilege principle*: processes should run with the minimum privileges required to accomplish their purpose. Some of these packages are: Linux capabilities [1], LIDS [23], Immunix SubDomain [16], LOMAC [21], VXE [30], Security-Enhanced Linux [29]. We used the Linux capabilities since they come with Linux and provide the functionality we need. Also, installing two patches in the same kernel (RT-Linux and the security package) can be complex and version dependent.

The TTCB builds on the Timely Computing Base work [43]. The objective of this distributed component it to assist the implementation of timed operations and to detect timing failures. It assumes a benign failure model, i.e., on the contrary to the TTCB it does not consider malicious faults. The TTCB provides not only all the functionality of the Timely Computing Base, but also additional security-related services.

The Agreement Service protocol is basically a Consensus protocol with the possibility of selecting the decision function and with the two additional masks in the result (see for instance [22]). Synchronous reliable multicast protocols are known for a long time [6, 7, 18]. Although these protocols assume a synchronous model, some are not timely. Also, they rely on diffusing the messages that are received to all the other recipients in order to guarantee reliability. The implementation of the Timely Computing Base gave us the insight that the network bandwidth is a very limited resource, if we want it to behave in real-time, so we designed our protocol without echos/diffusion (every message is broadcasted only once).

13 Conclusions and Future Work

The paper describes the design of a security kernel – the TTCB – with innovative features: first, it is distributed, with local parts in each node connected by a control channel; second, it is real-time capable of timely behavior; and third, it can be constructed using only COTS components. The paper also presents the services of the TTCB and gives an intuition on how these services can be used to support the construction of a new generation of timely and secure protocols. The first two protocols of this class are described in [15].

The current architecture of the TTCB is based on mainstream hardware running a real-time operating system, RT-Linux, and on a Fast-Ethernet network. By applying our design methodology, we expect that the existing implementation exhibits a good coverage of the assumptions, acceptable to most applications. This solution has one extra added advantage – the TTCB can be tested and used in open settings.

In the future, we are considering the implementation of the TTCB inside an appliance board. A version of RT-Linux for embedded systems is already available, which leads us to predict that the port will be straight forward. The second area of future work is in the development of new timed secure protocols.

Acknowledgements

The authors would like to thank António Casimiro, Pedro Martins, Nuno Miguel Neves and Lau Cheuk Lung for valuable discussions on several issues of this paper.

This work was partially supported by the EC, through project IST-1999-11583 (MAFTIA), and by the FCT, through the Large-Scale Informatic Systems Laboratory (LASIGE) and the project POSI/1999/CHS/33996 (DEFEATS).

References

[1] Linux Capabilities FAQ 0.2. ftp://ftp.guardian.no/pub/free/linux/capabilities/capfaq.txt, 2000.

[2] L. Alvisi, D. Malkhi, E. Pierce, M. K. Reiter, and R. N. Wright. Dynamic byzantine quorum systems. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, June 2000.

[3] S. Ames, M. Gasser Jr., and R. Schell. Security kernel design and implementation: An introduction. *IEEE Computer*, 16(7):14–22, 1983.

[4] Y. Amir, Y. Kim, C. Nita-Rotaru, J. Schultz, J. Stanton, and G. Tsudik. Exploring robustness in group key agreement. In *Proceedings of the 21th IEEE International Conference on Distributed Computing Systems*, April 2001.

[5] T. Aslam, I. Krsul, and E. Spafford. Use of a taxonomy of security faults. In *Proceedings of the 19th National Information Systems Security Conference*, October 1996.

[6] Ö. Babaoğlu and R. Drummond. Streets of Bizantium: Network architectures for fast reliable broadcasts. *IEEE Transactions on Software Engineering*, 11(6):546–554, April 1985.

[7] Ö. Babaoğlu, R. Drummond, and P. Stephenson. The impact of communication network properties on reliable broadcast protocols. In *Proceedings of the 16th IEEE International Symposium on Fault-Tolerant Computing*, pages 212–217, July 1986.

[8] M. J. Bach. *The Design of the UNIX Operating System*. Prentice/Hall International, 1986.

[9] M. Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, June 1997.

[10] M. Bishop and M. Dilger. Checking for race conditions in file access. *Computing Systems*, 9(2):131–152, 1996.

[11] A. Casimiro and P. Veríssimo. Timing failure detection with a Timely Computing Base. In *Proceedings of the European Research Seminar on Advances in Distributed Systems*, April 1999.

[12] A. Casimiro and P. Veríssimo. Using the Timely Computing Base for dependable QoS adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, October 2001.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[14] S. Cesare. Runtime kernel kmem patching. http://www.big.net.au/˜ silvio/, November 1998.

[15] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo. Efficient byzantine-resilient reliable multicast on a hybrid failure model. Submitted to DSN2002.

[16] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious server security. In *Proceedings of the USENIX 14th Systems Administration Conference*, December 2000.

[17] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, January 1998.

[18] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proceedings of the 15th IEEE International Symposium on Fault-Tolerant Computing*, pages 200–206, 1985.

[19] B. Dutertre, H. Saidi, and V. Stavridou. Intrusion-tolerant group management in Enclaves. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, June 2001.

[20] D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. Network Working Group, Request for Comments: 1750, December 1994.

[21] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2000.

[22] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[23] X. Huagang. Build a secure system with LIDS. http://www.lids.org, October 2000.

[24] IEEE. *Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - Amendment: Protection, Audit and Control Interfaces. P1003.1e.* March 1999. (withdrawn).

[25] N. Itoi and P. Honeyman. Smartcard integration with Kerberos v5. In *Proceedings of the USENIX Workshop on Smartcard Technology*, May 1999.

[26] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, volume 3, pages 317–326, January 1998.

[27] B. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.

[28] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws, with examples. Technical Report NRL/FR/5542–93-9591, Naval Research Laboratory, November 1993.

[29] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. http://www.nsa.gov/selinux/slinux-abs.html, April 2001.

[30] S. Lozovsky. VXE, a Linux security tool. LinuxFocus.org, January 2000.

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

[32] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*, July 1987.

[33] N. F. Neves and P. Veríssimo, editors. *First Specification of APIs and Protocols for MAFTIA Middleware. Project MAFTIA IST-1999-11583 deliverable D24.* August 2001.

[34] D. Powell, D. Seaton, G. Bonn, P. Veríssimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing*, June 1988.

[35] D. Powell and R. J. Stroud, editors. *MAFTIA: Conceptual Model and Architecture. Project MAFTIA IST-1999-11583 deliverable D2.* November 2001.

[36] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

[37] M. K. Reiter, K. P. Birman, and R. van Rennesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, November 1994.

[38] V. Shoup and A. D. Rubin. Session key distribution using smart cards. In Springer-Verlag, editor, *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques, Eurocrypt'96*, volume 1070 of *Lecture Notes in Computer Science*, May 1996.

[39] T. Stabell-Kulø, R. Arild, and P. H. Myrvang. Providing authentication to messages signed with a smart card in hostile environments. In *Proceedings of the USENIX Workshop on Smartcard Technology*, May 1999.

[40] Trusted Computing Platform Alliance (TCPA). Main specification version 1.1a. Technical report, TCPA, December 2001. http://www.trustedpc.org/.

[41] J. D. Tygar and B. S. Yee. Dyad: A system for using physically secure coprocessors. In *Proceedings of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment*, April 1993.

[42] U.S. Department of Defense. Trusted computer systems evaluation criteria, August 1983.

[43] P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, June 2000.

[44] P. Veríssimo, N. F. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the IEEE Third Information Survivability Workshop*, October 2000.

[45] P. Veríssimo, L. Rodrigues, and A. Casimiro. CesiumSpray: a precise and accurate global time service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294, May 1997.

[46] V. Yodaiken and M. Barabanov. RTLinux version two. http://www.fsmlabs.com/archive/design.pdf, 1999.

A Proofs of the Protocols

This appendix gives concise proofs of theorems and lemmas about the Local Authentication Service, the Agreement Service and Timely Reliable Broadcast protocols.

A.1 Local Authentication Service Protocol

This section is about the Local Authentication Service protocol in Figure 3.

Theorem 1 *The Local Authentication Service protocol is an authenticated key establishment protocol.*

Proof sketch. The protocol is an authenticated key establishment protocol if it verifies SK1 through SK4.

SK1. The protocol verifies SK1 since the key is passed only in the first message and only the local TTCB has the private key that can decrypt this message, $\langle E_u(K_{et}, X_e)\rangle$.

SK2. The proof of SK2 can be divided in two parts. The entity knows that the local TTCB has the key because it gave it (first message) and receives back a confirmation that cannot be falsified since it is obtained with the local TTCB private key (second message). The local TTCB knows that the entity has the key since it was the entity that gave the key to the local TTCB (first message).

SK3. is also verified since the entity gives the TTCB a message that only the TTCB can decrypt, $\langle E_u(K_{et}, X_e)\rangle$, and the TTCB gives back the decrypted challenge, X_e, showing that it was able to do the decryption.

SK4. Property SK4 primarily depends on the key generation method and encryption algorithm. We assume that the entity or the TTCB generate keys that verify that property. SK4 is guaranteed by the protocol since a key does not depend on a previous one. □

A.2 Agreement Service Protocol

This section is about the Agreement Service protocol in Figure 4.

Theorem 2 *If the Agreement Service is implemented with the Agreement Service protocol and there are no local TTCB crashes then it verifies Termination, Integrity, Agreement and Validity.*

Proof sketch. All local TTCBs broadcast the values proposed locally to all others. All receive and process the same values since there are no crashes and the protocol tolerates omissions in the network. The result is obtained applying a deterministic function to the values received. Therefore, all local TTCBs obtain the same result.

Termination. A correct entity eventually calls *decide* to obtain the result of the agreement. Since all local TTCBs obtain the result, every correct entity eventually decides a result.

Integrity. A entity that calls decide more than once and obtains the result more than once, obtains always the same result. Therefore, every correct entity decides at most one value.

Agreement. All local TTCBs obtain the same result so all correct entities decide the same.

Validity. The result is obtained in the local TTCBs applying the function *decision* to the values proposed. If a correct entity decides, then it decides that value obtained by the local TTCBs. □

Theorem 3 *The Agreement Service protocol verifies Timeliness and $T_{agreement}$ is given by:*

$$T_{agreement} = T_s + WCET_{send} + T_{send} + T_r + WCET_{receive} + \pi \tag{4}$$

Proof sketch. $WCET_{send}$ and $WCET_{receive}$ are respectively the worst case execution times of the send and receive routines. Any value proposed after *tstart* is not accepted for the agreement (Line 2). After the value is introduced in sendTable, the broadcast routine takes less than T_s to start to run, so the message with the value will not be broadcasted after $(tstart + T_s + WCET_{send})$ (Lines 8-9). The message will take at most T_{send} to arrive to the local TTCBs and the receive routine may take at the most T_r to start to run. Then the message will take less than $WCET_{receive}$ to be received and processed (Lines 11-15). The factor π takes in account the local TTCB clocks de-synchronization that leads to a different assessment of *tstart* in different local TTCBs. Adding all these maximum delays, we have the Formula 4. Since there is a $T_{agreement}$ the protocol has the property of Timeliness. □

A.3 Agreement Service Protocol With Local TTCB Crashes

This section is about the crash-tolerant Agreement Service protocol and the Timely Reliable Broadcast protocol in Figure 5.

Lemma 1 *The Timely Reliable Broadcast protocol verifies the properties of Validity, Agreement and Integrity if there are no local TTCB crashes.*

Proof sketch. *Validity.* A correct (non-crashed) local TTCB receives the messages that it R-broadcasts. If it R-broadcast $M(s, n)$ then it receives $M(s, n)$. After $M(s, n)$ it R-broadcasts $M(s, n + 1)$, $M(s, n + 2)$, etc. After receiving $M(s, n)$, when it eventually receives $M(s, n_+)$ it R-delivers $M(s, n)$, $\forall_{n_+} : n_+ > n$.

Agreement. If a correct local TTCB R-delivers $M(s, n)$, then it received $M(s, n)$ and a message in one of the two conditions tested in Line 9. Since we are not considering crashes and we assume that omissions in the network are tolerated, all local TTCBs receive those two messages and R-deliver $M(s, n)$.

Integrity. The property of Integrity means that no spurious messages are R-delivered. This is a consequence of AN2 and AN8. □

Lemma 2 *The Timely Reliable Broadcast protocol tolerates a single local TTCB crash in an interval of time (not assuming AN6).*

Proof sketch. If a local TTCB crashes during the R-broadcast of $M(s, n+1)$, and no local TTCB receives it, no local TTCB R-delivers $M(s, n)$. If at least a single local TTCB receives $M(s, n+1)$ it R-delivers $M(s, n)$ (first condition on Line 9) and, in the next message that it R-broadcasts, it will say to all the other local TTCBs that it received $M(s, n+1)$ (*higherseqVector[s]=n+1*) and they will also R-deliver $M(s, n)$ (second condition in Line 28).

Now, let us show that the protocol does not tolerate two or more crashes. Suppose that the local TTCB s crashed and a single local TTCB received $M(s, n+1)$. Then, the local TTCB that received the message started to R-broadcast a message with *higherseqVector[s]=n+1*. If it also crashed during that R-broadcast, some local TTCBs could receive confirmations while other did not; this would cause some local TTCBs to R-deliver $M(s, n)$, while others would not. Therefore, the protocol tolerates a single local TTCB crash during a reliable broadcast. □

Theorem 4 *The Timely Reliable Broadcast protocol tolerates Bd local TTCB crashes (assuming AN6).*

Proof sketch. Consider that the local TTCB s starts R-broadcasting $M(s, n+1)$ and crashes. The worst case happens when just Bd local TTCBs receive the message. We want to prove that the protocol tolerates Bd crashes. Since the sender already crashed only $Bd-1$ local TTCBs can still crash. Therefore, of Bd local TTCBs that received the message at least one does not crash and sends *higherseqVector[s]=n+1* in the next message it R-broadcasts. All non-crashed TTCBs receive that R-broadcast and R-deliver $M(s, n)$. □

Theorem 5 *The Timely Reliable Broadcast protocol has the property of Timeliness and the constant $T_{broadcast}$ is given by:*

$$\begin{aligned} T_{broadcast} = \quad & (WCET_{send} + T_{send} + T_r + WCET_{receive}) + (T_s) + \\ & (T_s + WCET_{send} + T_{send} + T_r + WCET_{receive}) + \pi \end{aligned} \tag{5}$$

Proof sketch. The first component of this delay is the maximum time for the local TTCBs to receive the message R-broadcasted, $M(s, n)$ (first pair of brackets). The sender sends the next message, $M(s, n+1)$, T_s after $M(s, n)$. Therefore, the first two components of the formula give the maximum time for the local TTCBs to receive $M(s, n+1)$. The third component of the formula gives the extra time for the non-crashed local TTCBs to receive a message with $n+1$ associated with s and to R-deliver $M(s, n)$. If there is a constant $T_{broadcast}$ then the protocol has the property of Timeliness. □

Theorem 6 *For the crash-tolerant Agreement Service Protocol, $T_{agreement}$ is given by:*

$$T_{agreement} = T_s + T_{broadcast} \tag{6}$$

Proof sketch. After the last entity proposes or *tstart* expires, the Agreement Service takes at most a send period T_s to R-broadcast the last propose. Then, all local TTCBs take at most $T_{broadcast}$ to R-deliver the message. □

B Complete Agreement Service Protocol

This appendix shows the complete Agreement Service protocol that combines the protocols in Figures 4 and 5.

For each local TTCB

```
propose routine
when entity calls TTCB_propose(eid, elist, tstart, decision, value) do
    if (entity already proposed) or (eid $\notin$ elist) or (clock() $>$ tstart) then return error;
    insert (elist, tstart, decision, eid, value) in sendTable;
    get $R \in$ dataTable : R.elist = elist $\wedge$ R.tstart = tstart $\wedge$ R.decision = decision;
    if (R = $\perp$) then R := (get_tag(), elist, tstart, decision, $\perp$); insert R in dataTable;
    return R.tag;
broadcast routine
when clock() = $round_s \times T_s$ do
    sender := my_id(); seq := $round_s$;
    M := (sender, seq, higherseqVector, sendTable);
    repeat $Od + 1$ times do broadcast(M);
    sendTable := $\perp$; $round_s := round_s + 1$;
receive routine
when clock() = $round_r \times T_r$ do
    while (read(M) $\neq$ error) do
        foreach M-ndlv in notDelivered do
            if [(M-ndlv.sender = M.sender) and (M-ndlv.number $<$ M.number)] or
                    (M-ndlv.number $<$ M.higherseqVector[M-ndlv.sender]) then
                        R-deliver(M-ndlv.sendTable); remove M-ndlv from notDelivered;
        if (higherseqVector[M.sender] $>$ M.number) then R-deliver(M.sendTable);
            else put M in notDelivered; higherseqVector[M.sender] := M.number;
    $round_r := round_r + 1$;
R-deliver routine (sendTable-in)
foreach (elist, tstart, decision, eid, value) $\in$ sendTable-in do
    get $R \in$ dataTable : R.elist = elist $\wedge$ R.tstart = tstart $\wedge$ R.decision = decision;
    if (R = $\perp$) then R := (get_tag(), elist, tstart, decision, $\perp$); insert R in dataTable;
    insert value in R.vtable;
    return;
decide routine
when entity calls TTCB_decide(tag) do
    get $R \in$ dataTable : R.tag = tag;
    if (R $\neq \perp$) and [(clock() $>$ R.tstart + $T_{agreement}$) or
        (all entities in non-crashed local TTCBs proposed a value)] then
            return (calculate result using function R.decision and values in R.vtable);
    else return error;
```

Figure 7: Crash-Tolerant Agreement Service internal protocol. Instance at a local TTCB.