The Design of a COTS Real-Time Distributed Security Kernel*

Miguel Correia, Paulo Veríssimo, and Nuno Ferreira Neves

Faculdade de Ciências da Universidade de Lisboa
Bloco C5, Campo Grande, 1749-016 Lisboa, Portugal
{mpc,pjv,nuno}@di.fc.ul.pt

Abstract. This paper 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

This paper describes the design of a security kernel called Trusted Timely Computing Base (TTCB). A security kernel [2] is a fail-controlled subsystem trusted to execute a few functions correctly, albeit immersed in an environment subjected to malicious faults. In the past, security kernels have mainly been used as *intrusion prevention devices*, by supporting the mediation/protection of all system interactions, and/or all accesses to system resources. The reference monitor paradigm is such an example [8]. Alternatively, we argue that a security kernel can be used as an *intrusion tolerance device.* The idea is to consider that most of the system runs in an environment prone to attacks, but there is a secure subsystem, the security kernel, that is used to run crucial phases of execution allowing a collection of entities to *tolerate* intrusions in some of them. Think for

* 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).

example in a web server with several replicas. A security kernel can run some steps of an intrusion tolerant protocol that provides correct results, even if some replicas are intruded and behave maliciously (i.e., try to break the protocol). Intrusion tolerance is the approach taken in the MAFTIA project, under which the TTCB is being developed [15] [1]. The TTCB assists the implementation of some of the intrusion-tolerant middleware components, whose architecture and general design principles have been described elsewhere [20]. The formal verification of the TTCB is on-going work in the context of the MAFTIA project, and will be the subject of future reports.

The TTCB has some innovative features. Firstly, it is a *distributed* subsystem with its own secure channel/network – the control channel/network (see Figure 1). A distributed security kernel represents a "hard-core" component, offering trusted services to a collection of participants [2], despite the fact that the latter reside in different nodes, and that their normal communication is through an insecure network – the payload network (see figure). 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 if 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 of distributed operations, and detecting timing failures.

Thirdly, the TTCB can be implemented using only COTS components, hardware and operating system. 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, a prototype of the TTCB that runs in mainstream PCs with RT-Linux, a real-time brand of Linux [3], is currently available for free non-commercial use.

The paper discusses essentially three things about our distributed security kernel. First, it presents the TTCB model (Section 3), describes the services provided by the TTCB and their implementation, with special emphasis on the security services (Section 4). Next it shows how resilience to intruders can be enforced in the proposed COTS-based implementation. The TTCB follows a design methodology based on a *composite fault model*, that clearly identifies the

[1] More information is available at the sites www.navigators.di.fc.ul.pt and www.maftia.org

[2] Throughout the paper we use interchangeably the words *entity* and *participant* to denominate any software component that uses the TTCB services.

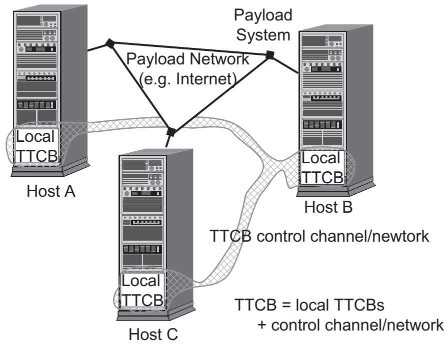

Fig. 1. The architecture of a system with a TTCB

malicious faults that have to be processed in order to prevent intrusions in the TTCB (Section 5). To conclude, the paper motivates the design of intrusion tolerant systems using the TTCB (Section 6).

2 Related Work

The TTCB is a distributed security kernel which is radically different from the classic Trusted Computing Base (TCB) [11] or the Network Trusted Computing Base (NTCB), composed by a set of interconnected TCBs [12]. The objective of both the TCB and the NTCB is to provide *intrusion prevention* for all critical software in the host, i.e., to prevent that attacks against a host have success. The TTCB, on the contrary, is supposed to be the only secure component of a host, and to provide a set of simple services that assist processes (or other software components) to tolerate attacks. Even if some processes are attacked with success, the TTCB assists the collection of processes to go on delivering their service correctly. In another paper we show how the TTCB can be used to execute intrusion tolerant protocols [4]. We are not aware of any distributed security kernel with the above mentioned characteristics of the TTCB. We are also not aware of any real-time security kernel.

The TTCB builds on the Timely Computing Base work [19]. 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 is not resilient to malicious faults. The TTCB provides not only all the functionality of the Timely Computing Base, but also additional security-related services.

There is some other work on the design of secure devices to assist the execution of secure applications. The Trusted Computing Platform Alliance (TCPA) is defining a secure *subsystem* that provides some local security services to applications, e.g., persistent storage and platform authentication [17]. Project Dyad explored the use of the Citadel secure coprocessor to implement a number of secure distributed applications, e.g., electronic payment [18]. Several papers describe the use of SmartCards with the same generic purpose [7, 16]. Although this paper describes the implementation of the TTCB in COTS PCs and OS, it could also be implemented inside devices like secure coprocessors or Smart-Cards. Moreover, the TTCB is a distributed component and therefore it can support and assist distributed applications in a more effective way. In fact, the protocols proposed in [4] can tolerate any number of faulty participants, which is an exciting result. The TTCB is also real-time, so it can assist the execution of applications with time requirements.

3 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* or *control network*, depending on the implementation. This set up of local TTCBs interconnected by the control channel/network is collectively called *the* TTCB. The TTCB is used to assist protocols/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, tamper-proof 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 direction followed was the second, the one based on COTS components (hardware and software). This design of the TTCB is discussed later in the paper.

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 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 [13].

4 TTCB Services

This section presents the TTCB security-related services and their design. The design is generic since it relies on an abstraction of the control network (Table 1).

AN1 Broadcast – The AN has an unreliable packet broadcast primitive
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 (AN) properties.

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, data block.
Trusted random numbers	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 an operation execution.
Trusted timing failure detection	Checks if an operation is executed in a time interval.
Trusted absolute timestamping	Provides globally meaningful timestamps.

Table 2. TTCB Services

4.1 TTCB Local Security Services

This section describes the local (non-distributed) security-related services of the TTCB, Local Authentication Service and Random Number Generation Service.

Local Authentication Service The purpose of this 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 [9]. 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 a trusted 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.

The protocol to establish the shared key has to be an *authenticated key establishment protocol* with local TTCB authentication. The protocol is presented in Figure 2. The formal properties of the protocol and the proof that it verifies those properties can be found in [5].

		Action	Description
1	P $\to$ 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 $\to$ P	$\langle S_r(X_e)\rangle$	TTCB sends the entity the signature of the challenge obtained with its private key K_r

Fig. 2. Local Authentication Service protocol

The shared key K_{et} has to be generated by the entity, not by the TTCB. We would desire it to be the other way around, but the only key they share initially is the local TTCB public key, that can be used by the entity to protect information that can be read only by the local TTCB (that has the corresponding private key) but not the contrary. K_{et} has to be generated by the entity 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 solve, however, a set of practical criteria can help to mitigate it: (1) the entity should use as much as possible sources of random data not controlled by the OS. (2) The entity should use as many different sources of random data as possible. Even if an intruder 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. (3) The entity must use a *strong mixing function*, i.e., a function that produces an output whose bits are uncorrelated to the input bits [6]. An example is a hash function such as MD4 or MD5. For similar reasons, the protocol challenge, X_e, has to be generated by the entity using the same approach.

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

`eid, chlg_sign` $\leftarrow$ **TTCB_localAuthentication**`(key, protection, challenge)`

The input parameters are the key, the communication protection to be used, and the challenge. The output parameters are the entity identification –*eid*– used to identify the entity in the subsequent calls, and the signature of the challenge.

Random Number Generation Service This service supplies uniformly distributed random numbers, which can be used as nonces or keys for cryptographic primitives such as distributed authentication protocols. The TTCB provides this service for efficiency since the method described in the previous section can be slow.

The interface of the service is a single function that returns a random number:

`number` ← **TTCB_getRandom**()

In a future version of the TTCB, based on an appliance board, we envisage the use of a hardware random number generator. In the current RT-Linux TTCB, the random numbers are given by the Linux random number generator. This generator 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.

4.2 TTCB Distributed Security Service

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 (see [13]).

The Trusted Block Agreement Service This service (Agreement Service for short) performs agreement protocols between sets of entities. These protocols, which for instance, multicast a number of bytes or reach to a consensus with a majority decision, are executed in a secure and timely fashion since the service runs inside the TTCB. The service is not intended 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*. An entity *proposes a value* when it calls *TTCB_propose*. An entity *decides a result* when it calls *TTCB_decide* and receives back a result. The function *decision* calculates the result in terms of the inputs of the service. Formally, the Agreement Service is defined by the following properties:

- *AS1 Termination.* Every correct entity eventually decides a result.
- *AS2 Integrity.* Every correct entity decides at most one result.
- *AS3 Agreement.* If a correct entity decides *result*, then all correct entities eventually decide *result*.
- *AS4 Validity.* If a correct entity decides *result* then *result* is obtained applying the function *decision* to the values proposed.
- *AS5 Timeliness.* Given an instant *tstart* and a known constant $T_{agreement}$, a process can decide by $tstart + T_{agreement}$.

The TTCB is a timely component in a payload system with uncertain timeliness. Therefore, the Timeliness property is 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: 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)
```

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*, which of the two happens first. 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 proposal after *tstart* it returns an error.

The other parameters of *TTCB_propose* are: *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; (4) *proposed-any*, a similar mask that indicates which entities proposed any value. Two *decision* functions currently available are: *TTCB_TBA_RMULTICAST*, that returns the value proposed by the first entity in *elist* (therefore the service works as a reliable multicast); *TTCB_TBA_MAJORITY*, that returns the most proposed value. Both return the two masks.

Trusted Block Agreement Service Protocol The internal protocol that implements the Agreement Service is time-triggered: *TTCB_propose* is called asynchronously, and gives the TTCB data that is stored in tables; periodically that data is broadcast to all local TTCBs, including the sender, and, also periodically, data is read from the network and processed.

The protocol uses two tables (Figure 3). 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 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.

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 by 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 so related to the algorithm functionality. If the propose is accepted, its data is inserted

```
                         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 ∉ elist) or (clock() > tstart) return error;
3    insert (elist, tstart, decision, eid, value) in sendTable;
4    get R ∈ dataTable : R.elist = elist ∧ R.tstart = tstart ∧ R.decision = decision;
5    if (R = ⊥) R := (get_tag(), elist, tstart, decision, ⊥); insert R in dataTable;
6    return R.tag;
broadcast routine
7  when clock() = round_s × T_s do
8    repeat Od + 1 times do broadcast(sendTable);
9    sendTable := ⊥; round_s := round_s + 1;
receive routine
10 when clock() = round_r × T_r do
11   while (read(M) ≠ error) do
12     foreach (elist, tstart, decision, eid, value) ∈ M.sendTable do
13       get R∈dataTable : R.elist = elist ∧ R.tstart = tstart ∧ R.decision = decision;
14       if (R = ⊥) R := (get_tag(), elist, tstart, decision, ⊥); insert R in dataTable;
15       insert value in R.vtable;
16   round_r := round_r + 1;
decide routine
17 when entity calls TTCB_decide(eid, tag) do
18   get R ∈ dataTable : R.tag = tag;
19   if (R≠⊥) and [(clock()>R.tstart+T_agreement) or (all entities proposed a value)]
20     return (calculate result using function R.decision and values in R.vtable);
21   else return error;
```

Fig. 3. Agreement Service internal protocol. Instance at a local TTCB.

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.

The above protocol can be proved correct if communication is done using a reliable broadcast primitive. This primitive and the proof can be found in [5].

5 The TTCB Design: Enforcing Resilience to Intruders

A system design addresses both functional and non-functional aspects. The functional aspects are concerned with the algorithms and protocols that make the system perform its service, mostly presented in the previous section. This section is concerned with the non-functional design of the COTS-based TTCB.

5.1 Design Methodology

Composite fault model with hybrid failure assumptions The organization of assumptions in terms of a composite fault model [20] 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 4. 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).

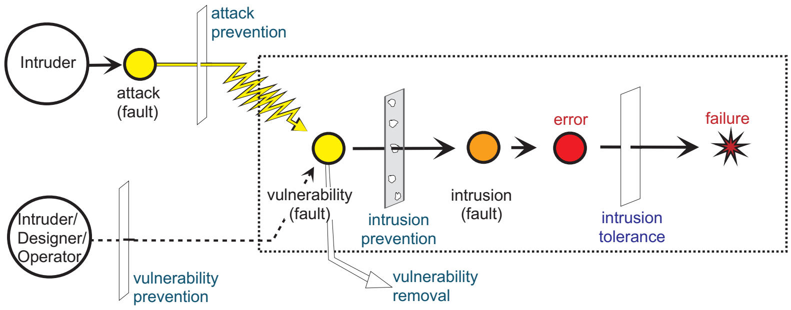

Fig. 4. The Composite Fault Model of MAFTIA

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.

Note that there is a body of research on *hybrid failures* for consensus and diagnosis algorithms, assuming failure type distributions for different participants [10, 22]. For instance, some participants are assumed to behave arbitrarily while others are assumed to fail only by crashing. The present work might best be described as *architectural hybridization*, in the lines of works such as [14, 21], where failure assumptions are in fact enforced by the architecture and construction of the system components, and thus well-founded. Hybrid behavior occurs component-wise: components in general are assumed to fail arbitrarily, but can use the services of a fail-controlled component, the TTCB.

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

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

Step one is the definition of the TTCB architecture and failure modes. The TTCB architecture was presented in Section 3 but is more detailed below in Section 5.2. The architecture itself can prevent some attacks against specific components. For example, the control network being physically inaccessible to hackers. About the failure modes, recall that we consider the local TTCBs to be fail-silent, and consider the inter-TTCB communication also to be fail-silent.

Step two is about the system's *environment*, i.e., about whatever is external to the system but that interacts with it: host hardware and OS, networks, intruders, etc. The environment is characterized in terms of a set of assumptions that, in practice, have to be enforced using adaptation mechanisms. The environment assumptions and the adaptation mechanisms are presented in the section 5.3.

Step three deals with constructing the mechanisms and protocols which enforce the fail-silent behavior of the TTCB, on the assumed environment and architecture. This resumes to make the TTCB resilient to attacks and intrusions. The design methodology may recursively be applied to the internal components of the TTCB as part of this step. This is discussed in Section 5.4.

Step four 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 and assessment of the TTCB design is on-going work on the context of project MAFTIA.

5.2 System Architecture

The general architecture of the TTCB was presented in Sections 1 and 3. It was also mentioned that our current implementation is based on common PCs with RT-Linux. To pursue the COTS strategy, our implementation is based on Fast-Ethernet, for campus-wide systems: we provide each host having a TTCB with an extra LAN adapter. We envisage future designs based on tamperproof hardware and wide-area networks such as an ISDN Virtual Private Network (VPN) [3]. A VPN provides a private channel, if we assume that the public telecommunications network is not eavesdropped. Additional security can be obtained using secure channels, e.g., encrypting the TTCB communication.

RT-Linux is an engineering of Linux, which was modified in order that a real-time executive takes control of the hardware, 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.

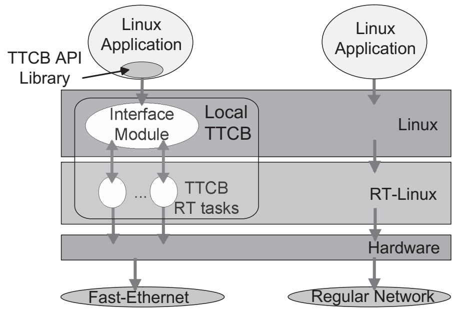

Fig. 5. The architecture of a host for the COTS-based TTCB

The COTS-based *local TTCB* architecture is detailed in Figure 5. The API functions are defined in libraries and communicate with the local TTCB using RT-Linux FIFOs. Currently there is one library for applications in C and another for Java (TTCB API Library in the figure). The local TTCB is implemented by an LKM (Interface Module) and by a number of RT tasks (TTCB RT Tasks). The TTCB Interface Module handles calls from the entities. It 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.

5.3 Environment Assumptions and Adaptation Mechanisms

The environment assumptions are shown in Table 3. The environment includes the PCs with RT-Linux (the *host*), the payload network and the control network.

RT-Linux and protection From the point of view of security, RT-Linux is very similar to Linux. Its main vulnerability is the ability a superuser has to control any resource in the system. This vulnerability is usually reasonably easy

[3] ISDN is a public digital network technology for data and telephony that provides connections with guaranteed bandwidth in multiples of 64 Kbps (128 Kbps, 1 Mbps. . .).

A1	The host protection mechanisms are not reconfigured by any intruder.
A2	The host kernel memory is not read or written by any intruder.
A3	The control channel access point is not read or written by any intruder.
A4	The data on the control channel is not read or written by any intruder.
A5	Given a known interval of time, the control channel does not corrupt more than k packets.
A6	There are no partitions in the control channel.

Table 3. Environment assumptions.

to exploit, e.g., using race conditions. Recently, several Linux extensions try to compartmentalize the power of the superuser. Linux *capabilities* [1], already part of the kernel, are privileges or access control lists associated with processes, allowing a fine grain control on how they use certain objects. Currently, though, the practical way of using this mechanism is quite basic. There is a system wide *capability bounding set* that bounds the capabilities that can be held by any system process. Removing a capability from that set disables the ability to use an object. Although basic, this mechanism fits our needs.

Enforcing environment assumptions Assumptions A1 and A2 impose the only limits on what the intruder can do inside a host. Otherwise, we assume that it can access the host, run software there, and become root or run processes with superuser privileges.

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 [4].

Assumption A2 protects the working space of both the RT-Linux kernel and the modules that support the TTCB. If the intruder 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:

- *Loadable kernel modules insertion:* LKMs allow a hacker that gained superuser privileges to insert code in the kernel. The vulnerability is disabled removing the capability `CAP_SYS_MODULE` off the capability bounding set.
- `/dev/mem` *and* `/dev/kmem` *devices:* these devices can be used to read and modify the system memory, including the kernel. This vulnerability is removed taking `CAP_SYS_RAWIO` off the capability bounding set.

Assumptions A3 through A6 refer to the control channel. Assumption A3 stipulates that an intruder cannot access the control network adapter from inside the host, and in consequence, he/she can neither send to, nor read and/or

[4] This discussion concerns the environment during runtime. We also assume that the kernel and the local TTCB binaries are not corrupted by an intruder or, at least, corruption is detected during reboot.

intercept packets from, the control network. This can be enforced modifying the relevant LAN controller.

Assumption A4, on the other hand, is secured by ensuring that an intruder does not 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 intruder 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, that are hard for the common Internet intruder to tamper with in conjunction with an attack through the payload network. Note however that assumption A4 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 A5 and A6 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 A6 in wider-area networks.

5.4 Enforcing System Failure Modes

The composite fault model in Figure 4 shows that different techniques can be used to make a system resilient to intrusions. The *intruder* in the figure is part of the environment, so its behavior is modelled by the environment assumptions in Table 3. Now, look at assumptions A1 through A4 in the table: they impose restrictions to the behavior of the intruder. Hypothesizing about limits to the behavior of malicious entities, such as hackers or viruses is, of course, not acceptable. Therefore, in the previous section we devised mechanisms that impose these restrictions in practice, i.e., that enforce the assumptions despite the potential arbitrary behavior of the intruder.

Assumptions A1 through A4 effectively do attack prevention (see Figure 4): it is an assumption that the intruder is not able to attack either the TTCB software modules or the control channel. Therefore, at this stage there is no need to enforce the system resilience to intruders. Handling the attacks/intrusions at step two (environment assumptions) is the same as doing it at step three. If we made RT-Linux part of the system then it would be the system that would be preventing or tolerating the faults, instead of the environment, i.e., protection would be made in step three instead of two. However, in this particular case, the way it is done seems more intuitive.

What remains to be defined at this stage is how the abstract network properties (Table 1) are obtained on top of the real network, taking in account the environment assumptions.

Property AN1 is available in the Ethernet and can be simulated with IP multicast or with several message 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 (assumptions A3/A4). If there are attacks, message integrity checks (MICs) can be used instead. Property AN3 is guaranteed by the environment assumption A5. 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 that: (1) only one host can be connected to each switch port (hubs cannot be used); and (2) the traffic load has to be controlled. The first requirement is obvious. The second is solved by an access control mechanism, that accepts or rejects the execution of a service taking in account the availability of resources (buffers and bandwidth). Property AN5 is guaranteed by the assumption A6. For property AN6, in a switched Fast-Ethernet *Bd* can easily exceed half of the nodes. Properties AN7 and AN8 are guaranteed by the assumptions A3 and A4 and could be enhanced using common cryptographic schemes.

6 Intrusion Tolerance with the TTCB

After delving into the discussion of the TTCB services and design, 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.

6.1 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. Hybrid assumptions, combining different kinds of failure assumptions, are followed in our work. This is because controlled failure assumptions have the problem of coverage in case of malicious faults, and arbitrary failure assumptions, on the other hand, are costly in terms of timeliness and complexity. 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. However, such an approach is only feasible when the fault model is well-founded, otherwise the system becomes easy prey to hackers. In consequence, the implementation of the TTCB discussed in Section 5 combines different techniques and methods tackling different classes of faults, in order to achieve the postulated behavior (fail-silent) with high coverage.

6.2 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 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.

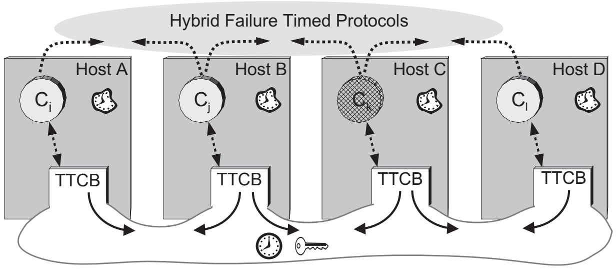

Fig. 6. Intrusion Tolerance with a TTCB

Observe Figure 6: 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 color is used to mean a trusted environment (the TTCB). The key means the environment is cryptographically secure. The grey colors for the payload system mean untrusted.

Trusting the TTCB security kernel means that 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 [4], where we give a practical example of the use of the TTCB to implement intrusion-tolerant protocols.

In order to understand the assumptions on timeliness of our system, let us analyze Figure 6 again: 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.

Constructing secure timed protocols in these environments is a hard 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. However, certain services, if provided in a trusted way (by the TTCB, which has thus to be a synchronous— real-time— component) can provide invaluable help.

6.3 Example Applications with a TTCB

This section exemplifies the use of the TTCB in two different settings. Figure 7(a) shows a web server replicated inside a facility, a company or another institution.

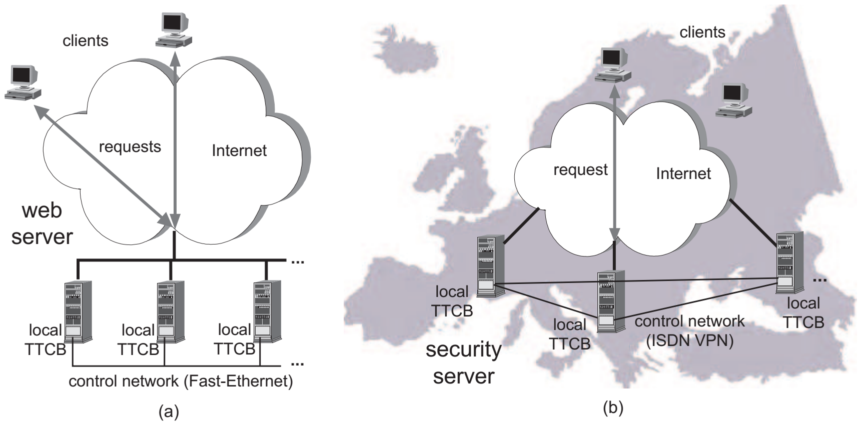

Fig. 7. Examples of intrusion tolerant systems with a TTCB: (a) replicated web server; (b) distributed security server

Clients call the server using an intrusion tolerant protocol. This protocol uses the TTCB to perform some crucial steps, but otherwise runs in the payload system. If a subset of replicas is corrupted and behave maliciously, the server will still provide correct results, tolerating these malicious faults. An inside-facility TTCB can be the version described in the paper. This solution requires an extra Fast-Ethernet adapter per host and an extra network switch, an affordable price.

Several Internet authentication schemes rely on highly secure and distributed servers. For instance, Public Key Infrastructures have Certification Authorities (CAs) with these characteristics. Figure 7(b) shows a TTCB distributed over a wide area, that allows the execution of intrusion tolerant protocols over such an extension. The TTCB control channel has to be a highly secure and wide channel, with guaranteed bandwidth (e.g., the above mentioned ISDN VPN).

7 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 hosts 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 intrusion tolerant protocols [4]. The currently available implementation of the TTCB is based on common 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.

References

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

2. S. Ames, M. Gasser Jr., and R. Schell. Security kernel design and implementation: An introduction. *IEEE Computer*, 16(7):14–22, 1983.
3. M. Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, June 1997.
4. M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo. Efficient byzantine-resilient reliable multicast on a hybrid failure model. In *Proc. of the 21th IEEE Symposium on Reliable Distributed Systems*, October 2002.
5. M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel (extended version). DI/FCUL TR 01–12, Department of Computer Science, University of Lisbon, 2001.
6. D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. IETF Network Working Group, Request for Comments 1750, December 1994.
7. N. Itoi and P. Honeyman. Smartcard integration with Kerberos v5. In *Proc. of the USENIX Workshop on Smartcard Technology*, May 1999.
8. B. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.
9. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
10. F. Meyer and D. Pradhan. Consensus with dual failure modes. In *Proc. of the 17th IEEE International Symposium on Fault-Tolerant Computing*, July 1987.
11. National Computer Security Center. Trusted computer systems evaluation criteria, August 1983.
12. National Computer Security Center. Trusted network interpretation of the trusted computer system evaluation criteria, July 1987.
13. 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. http://www.research.ec.org/maftia/deliverables/d24final.pdf.
14. D. Powell, editor. *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. ESPRIT Research Reports. Springer-Verlag, November 1991.
15. D. Powell and R. J. Stroud, editors. *MAFTIA: Conceptual Model and Architecture. Project MAFTIA IST-1999-11583 deliverable D2.* November 2001. http://www.research.ec.org/maftia/deliverables/D2fin.pdf.
16. T. Stabell-Kulø, R. Arild, and P. H. Myrvang. Providing authentication to messages signed with a smart card in hostile environments. In *Proc. of the USENIX Workshop on Smartcard Technology*, May 1999.
17. Trusted Computing Platform Alliance (TCPA). Main specification version 1.1a. Technical report, TCPA, December 2001. http://www.trustedpc.org/.
18. J. D. Tygar and B. S. Yee. Dyad: A system for using physically secure coprocessors. In *Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment*, April 1993.
19. P. Veríssimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 533–542, June 2000.
20. P. Veríssimo, N. F. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proc. of the IEEE Third Information Survivability Workshop*, October 2000.
21. P. Veríssimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global clock service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294, 1997.
22. C. Walter, N. Suri, and M. Hugue. Continual on-line diagnosis of hybrid faults. In *Proc. of the 4th IFIP International Working Conference on Dependable Computing for Critical Applications*, 1994.