Intrusion-Resilient Middleware Design and Validation [*]

Paulo Verissimo Miguel Correia Nuno Neves Paulo Sousa

University of Lisboa, Faculty of Sciences, LaSIGE
Bloco C6, Campo Grande, 1749-016 Lisboa - Portugal

{pjv,mpc,nuno,pjsousa}@di.fc.ul.pt
http://www.navigators.di.fc.ul.pt

Abstract

Intrusion Tolerance has become a reference paradigm for dealing with intrusions and accidental faults, achieving security and dependability in an automatic way, much along the lines of classical fault tolerance. This chapter is an introduction to the design and validation of intrusion-tolerant middleware and systems.

1 Introduction

Intrusion Tolerance (InTol) is a new security and dependability paradigm that slowly emerged in the past two decades. While most security paradigms attempt to prevent intrusions from occurring, InTol assumes that systems are so complex that vulnerabilities are inevitable, therefore intrusions will happen and have to be tolerated. The approach is similar to classical fault tolerance (or dependability), in which systems are designed not only to prevent accidental faults from happening, but also acknowledging that they will inevitably happen and that the system has to tolerate them automatically.

Intrusion Tolerance has become a reference paradigm for dealing with faults and intrusions, achieving security and dependability in an automatic way, much along the lines of classical fault tolerance. The paradigm presents a significant added value in face of what are the current and future perceived threats to computer systems: it allows designers to address both faults and attacks in a seamless manner, through a common approach to security and dependability. InTol is bound not to replace but instead to amplify the reach of the classical paradigms in security, which have mostly consisted in trying to prevent security hazards from happening, or

[*]P. Verissimo, M. Correia, N. Neves, and P. Sousa are with the Department of Informatics of the University of Lisboa Faculty of Sciences. P. Verissimo is a professor. M. Correia, N. Neves, and P. Sousa are assistant professors.

in deploying ad-hoc countermeasures when incidents are detected. In contrast, in InTol it is assumed that: systems remain to some extent faulty and/or vulnerable; attacks on components can happen and some will be successful; but automatic mechanisms ensure that the overall system nevertheless remains secure and operational.

The usual way to deploy InTol mechanisms in Internet-like or distributed systems is through middleware layers or web services that are fault- and intrusion-tolerant, which can then be used by upper layer services and applications transparently, independently of how tolerance is achieved. The general idea is to implement the middleware offering the service through n replicas cooperating through distributed protocols. Replicas will be attacked and corrupted at the measure of the power of the attacker, but as long as there are sufficient replicas to perform the service correctly, the system continues to function, sometimes even without the user noticing anything.

Given the severity and the malicious intelligence behind the expect threats, there is a need for protocols to resist in general to arbitrary faults as the top level of severity, in the line of what is called Byzantine fault/intrusion tolerance. The necessary number of replicas varies with system configuration, the baseline being that if one expects a number f of faults or intrusions, then the middleware implementing the service should actually consist (typically) of at least $n = 3f + 1$ replicas.

However, these challenges are so intense that one must in practical cases resort to defences that attempt at shrinking the attackers' chances. Designers resort to representing the system along hybrid distributed systems models, which allow to include trusted-trustworthy components as enhancers of the baseline Byzantine algorithms, obtaining significantly better efficiency vs. security ratios.

Furthermore, since faults erode systems inexorably, the next step towards high resilience is to offer strong enough resistance to attacks in order to prevent replica exhaustion. That is, to endow systems with the capability of preserving the needed resources (replicas) to perform correctly, throughout their mission. In other words, with the capability of achieving what is called exhaustion-safety. For our InTol middleware, this would mean to always preserve the number of replicas above the minimum threshold. This is a difficult task, since in the malicious-fault plane threats can be exacerbated by factors such as attacker power and common-mode vulnerabilities. Techniques that make the life difficult to the attacker, e.g., by exploiting diversity, rejuvenation, have been employed.

Last but not least, it is necessary to study and understand how malicious faults such as attacks are produced and what their effect is on existing vulnerabilities, in order to validate the fault assumptions underlying the above-mentioned intrusion-tolerant algorithms. This will allow algorithm and system designers to introduce more realistic assumptions. We are still far from a thorough understanding of the mechanisms behind the trilogy attack-vulnerability-intrusion.

Outline of this chapter

Middleware Design Principles Architecting intrusion-tolerant systems, to arrive at some notion of *intrusion-tolerant middleware* for application support, presents multiple challenges. Intrusion tolerance mechanisms can be selectively used at all layers starting with hardware, to build layers of progressively more trusted components and middleware subsystems, from baseline untrusted components (hosts, networks). This leads to an automation of the process of building resilience: for example, at lower layers, basic intrusion tolerance mechanisms are used to construct a trustworthy communication subsystem. This subsystem can then be trusted by higher layer distributed software, to securely communicate amongst participants without bothering about network intrusion threats. Or alternatively, it can be used to build an even more trustworthy higher layer— by incrementally using intrusion tolerance mechanisms— such as a replication management protocol resilient to both network and host intrusions. The first section introduces these middleware design principles.

Tolerating Intrusions Intrusion-tolerant middleware is usually based on the notion of replication, i.e., in scattering a service in a set of server machines. This section starts by presenting the main intrusion tolerance paradigms in the literature and how they are used to ensure the integrity of a service, i.e., that the latter behaves as expected even if there are intrusions in some of its components (servers and/or clients). Then the section presents how these paradigms can be extended to ensure also the confidentiality of information stored in a replicated service. Finally, some system architectures are presented, showing how the paradigms can be used to build actual systems.

Resisting Attacks Intrusion-tolerant systems may have a long lifetime (e.g., online banking systems) and it should be guaranteed that no more than the maximum number of tolerated faults ever occurs, i.e., InTol systems should be exhaustion-safe. This section starts by presenting a model that allows assessing the exhaustion-safety of a system according to its fault and timing assumptions. Then, it is shown that, in order to meet the exhaustion-safety predicate, InTol systems should be designed with diversity in mind and enhanced with proactive and reactive recovery mechanisms targeting, respectively, stealth/dormant faults and conspicuous attacks. It is also explained how recoveries can eliminate the effects of faults/attacks and how they can randomize certain parts of the system in order that vulnerabilities are somehow changed or removed, and the adversary cannot make use of knowledge learnt before the recovery.

Testing Attacks and Vulnerabilities An intrusion can only occur if the system contains vulnerabilities which can be exploited by an attack. This section presents some of the techniques that can be utilized to locate security vulnerabilities and allow their subsequent elimination, both during the testing phases of the development cycle and when the system is in operation. The following techniques are examined: static vulnerability analyzers, fuzzing mechanisms, and attack injection. Vulnerability removal is important because it causes an increase on the effort necessary to compromise a machine. The section also explains how these techniques can be applied to experimentally validate some aspects of the attack model and fault assumptions made during the design of the intrusion-tolerant system. Both aspects contribute to the deployment of cheaper intrusion-tolerant systems.

2 Intrusion-Tolerant Middleware Design Principles

In this section, we start by introducing the main concepts behind intrusion tolerance. The reader may find a thorough treatment in [145]. Then we move on explaining how to architect and design intrusion-tolerant middleware.

2.1 Intrusion Tolerance in a Nutshell

What is Intrusion Tolerance? As said earlier, the tolerance paradigm in security: assumes that systems remain to a certain extent vulnerable; assumes that attacks on components or subsystems can happen and some will be successful; ensures that the overall system nevertheless remains secure and operational, with a quantifiable probability. In other words:

- faults— malicious and other— occur;
- they generate errors, i.e., component-level security compromises;
- error processing mechanisms make sure that security failure is prevented.

2.1.1 AVI composite fault model

The mechanisms of failure of a system or component, security-wise, range from internal faults (i.e., vulnerabilities), to external, interaction faults (i.e., attacks), whose combination produces faults (i.e., intrusions) that can directly lead to a security failure. Figure 1a represents the fundamental sequence of these three kinds of faults: attack $\rightarrow$ vulnerability $\rightarrow$ intrusion $\rightarrow$ failure. This well-defined relationship is called the *AVI composite fault model*.

Vulnerabilities are faults in a computing or communication system that can be exploited with malicious intention. They are the primordial faults existing inside the components, essentially requirements, specification, design or configuration faults (e.g., coding faults allowing program stack overflow, files with root setuid in UNIX, naive passwords, unprotected TCP/IP ports).

Attacks are malicious intentional faults attempted at a computing or communication system, with the intent of exploiting one or more of those vulnerabilities in the system (e.g., port scans, email viruses, malicious Java applets or ActiveX controls).

An *intrusion* is an intentionally malicious operational fault resulting from a successful attack on a vulnerability. An intrusion has thus two underlying causes, as seen in the Figure 1a: vulnerability; and attack.

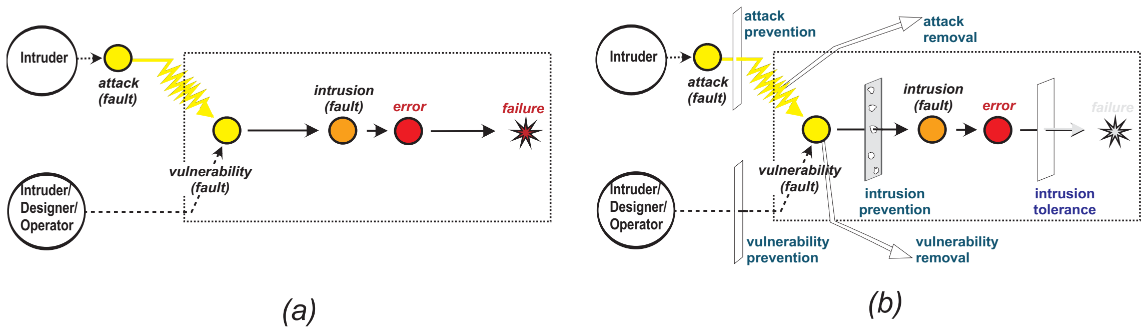


Figure 1: (a) AVI composite fault model; (b) Preventing security failure

2.1.2 Why the AVI model?

Firstly, it describes the mechanism of intrusion precisely: without matching attacks, a given vulnerability is harmless; without target vulnerabilities, an attack is irrelevant. Secondly, it provides constructive guidance to build in dependability against malicious faults, through the combined introduction of several techniques.

To begin with, we can prevent some attacks from occurring, reducing the level of threat, as shown in Figure 1b. *Attack prevention* can be performed, for example, by shadowing the password file in UNIX, making it unavailable to unauthorized readers, or filtering access to parts of the system (e.g., if a component is behind a firewall and cannot be accessed from the Internet, attack from there is prevented). We can also perform *attack removal*, which consists of taking measures to discontinue ongoing attacks.

However, it is impossible to prevent all attacks, so reducing the level of threat should be combined with reducing the degree of vulnerability, through *vulnerability prevention*, for example by using best-practices in the design and configuration of systems, or through *vulnerability removal* (i.e., debugging, patching, disabling modules, etc.) for example it is not possible to prevent the attack(s) that activate(s) a given vulnerability.

The whole of the above-mentioned techniques prefigures what we call *intrusion prevention*, i.e., the attempt to avoid the occurrence of intrusion faults. Figure 1b suggests, as we discussed earlier, that it is impossible or infeasible to guarantee perfect prevention. The reasons are obvious: it may be not possible to handle all attacks, possibly because not all are known or new ones may appear; it may not be possible to remove or prevent the introduction of new vulnerabilities.

For these intrusions still escaping the prevention process, forms of *intrusion tolerance* are required, as shown in the figure, in order to prevent system failure. As will be explained later, these can assume several forms: detection (e.g., of intruded account activity, of trojan horse activity); recovery (e.g., interception and neutralization of intruder activity); and/or masking (e.g., voting between several components, including a minority of intruded ones).

2.1.3 Trust and Trustworthiness

There is a well-defined relationship between the notions of "trust" and "trustworthiness"—in a sense, they relate strongly to the words "dependence" and "dependability" [1]. Whereas *trust* is the accepted dependence of a component or system, on a set of properties of another component or system, *trustworthiness* would be the measure in which the latter component or system meets that set of properties. Although this relation is often forgotten, leading designers to concentrate on only one of them at a time, it is crucial to the design of intrusion-tolerant systems.

The relation can be metaphorically described as *"Thou Shalt Not Trust non-Trustworthy Systems"* [142]. Let us understand this better. A trusted component has a set of properties that are relied upon by another component. If A trusts B, this means that a violation in those properties of B might compromise the correct operation of A. Observe that those properties of B trusted by A might not correspond quantitatively or qualitatively to B's actual properties. This happens when the relation is not taken into account by the designer. In consequence, trust should be placed *to the extent of* the component's trustworthiness. The trustworthiness of a component is thus, not surprisingly, defined by how well it secures a set of functional and non-functional properties, deriving from its architecture, construction, and environment, and evaluated as appropriate.

2.2 Models and assumptions

Surprising as it may seem, intrusion tolerance is not just another instantiation of accidental fault tolerance. Architecting intrusion-tolerant systems, to arrive at some notion of *intrusion-tolerant middleware* for application support, presents multiple challenges, primarily because several of the paradigms and models used in accidental fault tolerance are not adequate for malicious faults: potential for maliciously caused common-mode faults makes probabilistic assumptions risky (number of "independent" faulty components, fault types); error propagation is the rule rather than the exception (error detection delay, progressive intrusion); typical severity of malicious faults (Byzantine behaviour, attacks on timing, contamination of runtime support environment).

2.2.1 Failure assumptions

A crucial aspect of any fault-tolerant architecture is the fault model upon which the system architecture is conceived, and component interactions are defined. A system fault model is built on assumptions about the way system components fail. Classically, these assumptions fall into two kinds: *controlled failure* assumptions, and *arbitrary failure* assumptions.

Controlled failure assumptions specify constraints on component failures. For example, it may be assumed that components only fail by crashing or only have timing failures. This approach represents very well how common systems work under the presence of accidental faults, failing in a benign manner most of the time. However, it is difficult to model the behaviour of a hacker, so there is a problem of coverage that does not recommend this approach for malicious faults, unless a trustworthy solution can be found.

Arbitrary failure assumptions ideally specify no qualitative constraints on component failures. Practical systems do however specify quantitative bounds on the number of failed components. In this context, an arbitrary failure means the capability of generating a message at any time, with whatever syntax and semantics (form and meaning), and sending it to anywhere in the system. 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 failure assumptions combining both kinds of failures might be a way out of this dilemma [92]. Here, an undistinguished set of nodes are assumed to behave arbitrarily while others are assumed to fail only by crashing. These distributions are in essence postulating heterogeneous failure modes in sets of homogeneous components, a probabilistic foundation that makes sense in accidental fault scenarios, but which might be hard to sustain in the presence of malicious intelligence.

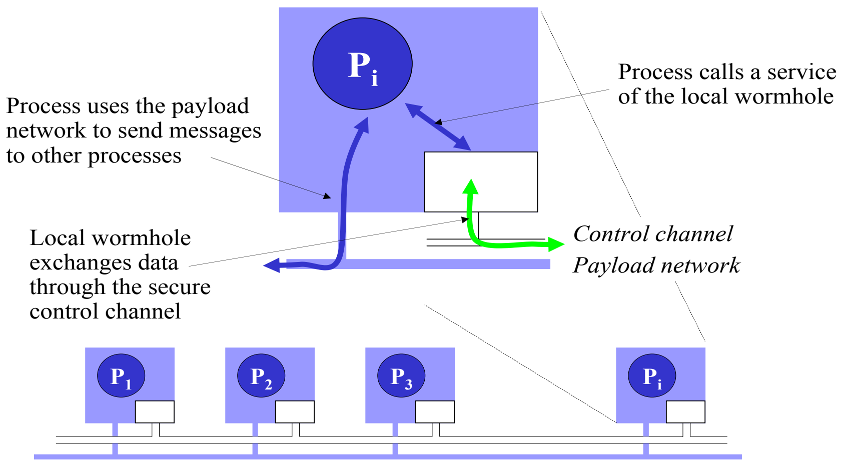
Architecturally hybrid failure assumptions introduce the necessary constraints [144]: some parts of the system are justifiably assumed to exhibit fail-controlled behaviour, whilst the remainder of the system is still allowed an arbitrary behaviour. This is an interesting approach for modular and distributed system architectures, especially intrusion-tolerant ones, so we develop the principle a bit further in the next section.

2.2.2 Intrusion tolerance under hybrid models

Consider a component or sub-system for which a given controlled failure assumption is made, the rest of the system being arbitrary on failure. How can we achieve coverage of such an assumption, given the unpredictability of attacks and the elusiveness of vulnerabilities?

A model featuring *architectural hybridization* is one where environment properties may vary from component to component. This includes failure assumptions, where the presence and severity of vulnerabilities, attacks and intrusions are in fact constrained by the architecture and the construction of those system components, and thus substantiated. For example, through intrusion prevention techniques combined with the recursive use of InTol mechanisms to build the component itself.

A modelling approach relevant to this discussion is the hybrid distributed systems model, or Wormholes model [143]. Figure 2 shows a possible representation of a system under this



P$_i$
Process calls a service of the local wormhole
Process uses the payload network to send messages to other processes
Local wormhole exchanges data through the secure control channel
Control channel
Payload network
P$_1$
P$_2$
P$_3$
P$_i$

Figure 2: Architecturally hybrid Wormholes system (payload displayed in dark and wormhole in white)

model. Under such a model, a system is a hybrid of a ‘normal’ or payload part, which can exhibit weak properties such as asynchrony and arbitrary or Byzantine failure, and a ‘privileged’ or control part, a subsystem capable of providing a small set of services with stronger properties (e.g., timeliness, security, crash failure) that are otherwise not available in the rest of the system. That latter part is also called a ‘wormhole’. Wormholes can be either local or distributed, in which case they are interconnected through a control channel (the case depicted in the figure). Practical wormholes should be kept small and simple in order to ensure their behaviour is verifiable.

Processes run in the payload, that part of the system that may experience arbitrary delays or failures (e.g., asynchronous Byzantine environment) and they communicate by sending messages through the payload network. However, during their execution, they can call the wormhole to perform (small) crucial steps. In contrast to the rest of the system, the wormhole always returns trustworthy results. Extremely resilient but performing secure protocols can be built with this powerful combination [29, 30].

The approach yields implementations of fault and intrusion-tolerant protocols that achieve the best of both worlds [141]: (i) more efficient than protocol implementations that have to deal with truly arbitrary failure assumptions for all components, and (ii) more robust than designs that make controlled failure assumptions without enforcing them.

Arbitrary failure assumptions considered necessary Note that the hybrid failure approach, no matter how resilient, relies on the coverage of the fail-controlled assumptions. Definitely, there will be a significant number of operations whose value and/or criticality is such that the risk of failure due to violation of these assumptions cannot be incurred. In consequence, an important area of research is still related to arbitrary-failure resilient building blocks, namely communication protocols of the Byzantine class, which do not make assumptions on the existence of trusted or controlled-failure components. They reason in terms of admitting any behaviour from the participants, and allow the corruption of a parameterizable number of participants, say f. The system works correctly as long as there exist $n > 3f$ participants. These protocols do not make assumptions about timeliness either, and are in essence time-free. This has implications on the operational aspects, such as performance.

2.3 Architectural notions

One key aspect of *intrusion-tolerant middleware* is that it should lead to an automation of the process of building resilience. In other words, each layer, starting with hardware, should contribute to building a next layer of more trusted functionality, progressively filtering the possible intrusions that may occur in the baseline untrusted components.

Additionally, a good design practice is for the middleware to be modular and tending to achieve incremental levels of trustworthiness. This is an accepted design principle for building distributed fault tolerance into systems. It facilitates the definition of different redundancy strategies for different components, and the placement of the relevant replicas.

For example, at lower layers, basic intrusion tolerance mechanisms are used to construct a trustworthy communication subsystem. This subsystem can then be trusted by higher layer distributed software, to securely communicate amongst participants without bothering about network intrusion threats. Or alternatively, it can be used to build an even more trustworthy higher layer— by incrementally using intrusion tolerance mechanisms— such as a replication management protocol resilient to both network and host intrusions.

As such, there are several relevant levels at which trust can be built, and the structure of an intrusion-tolerant host relies on a mix of a few architectural options:

- The notion of *trusted* — versus untrusted — *hardware*. As a good practice, most hardware should be considered to be untrusted, but small parts of it may be considered to be trusted, if adequately trustworthy, for example, by being *tamper-proof* by construction.

- The notion of *trusted support software*. Trusted to execute a few functions correctly albeit immersed in an environment subjected to malicious faults. The use of trusted hardware may help substantiate this assumption.

- The notion of *run-time environment*, extending operating system capabilities and hiding heterogeneity. A generic concept which is also useful in this context. Functions supplied by the above-mentioned trusted support software should be offered through the run-time API, vertically to all middleware layers.

- The notion of *trusted distributed component*. Implemented by each layer of the modular *middleware*: multipoint network abstraction, communication support services, and activity support services. At each level, the faulty behaviour of lower levels is partially or totally overcome. This depends on the different InTol strategies that can be followed at several levels of abstraction of the architecture. Versatile combinations of synchrony and failure assumptions are possible, from synchrony to asynchrony, from arbitrary to fail-silent.

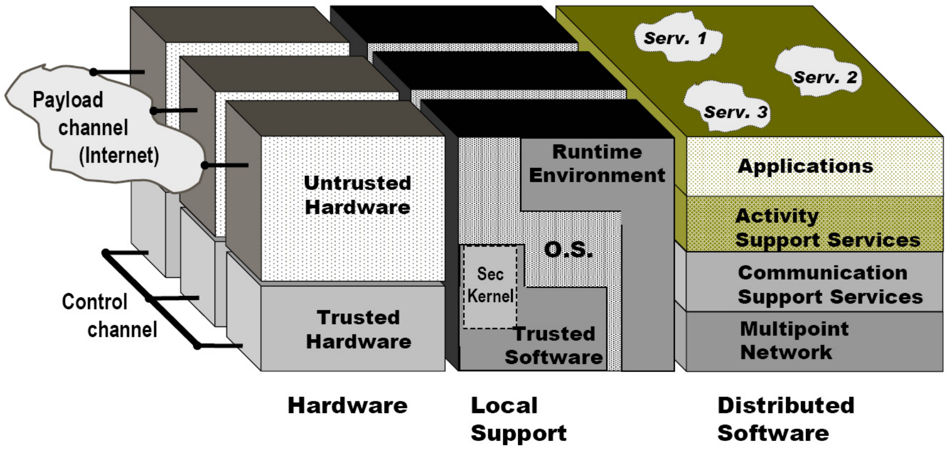


Figure 3: Dimensions of a reference intrusion-tolerant architecture

A reference intrusion-tolerant architecture inspired by the MAFTIA project work [144] is depicted in at least three different dimensions in Figure 3:

- First, there is the *hardware* dimension, which includes the host and networking devices that make up the physical distributed system.
- Second, within each node, there are the *local support* services provided by the operating system and the run-time platform. These may vary from host to host in a heterogeneous system, and some services may even not be available on some hosts or may have to be accessed via the network using protocols providing an appropriate degree of trust. However, at a minimum, the local services include typical operating system functionality such as the ability to run processes, send messages across the network, access local persistent storage (if it exists), etc.
- Third, there is the *distributed software* provided by the specific systems: the layers of *middleware*, running on top of the run-time support mechanisms provided by each host; and any system's native *services* (Serv. x in the figure).

Applications built to run on top of such an architecture use the abstractions provided by the middleware and the application services to operate securely across several hosts, and/or be accessed securely by users running on remote nodes, even in the presence of malicious faults.

As mentioned earlier, a middleware layer may host a trusted distributed component that overcomes the fault severity of lower layers and provides certain functions in a trustworthy way. These are in turn trusted by the layers above, in a recursive way. For example, a (distributed) transactional service trusts that a (distributed) atomic multicast component ensures the typical properties (agreement and total order), regardless of the fact that the underlying environment may suffer Byzantine malicious attacks.

The distribution dimension impacts on the protocol design but not on the services provided by each host. These are constructed on the functionality provided by the several middleware modules, represented in Figure 3:

- The lowest layer is the *Multipoint Network* module, *MN*, created over the physical infrastructure. This component of the reference architecture should hide the particularities of the underlying network to which hosts are directly attached. It should provide a run-time compliant interface for any standard protocols to be used (e.g., IP, IPSEC, SNMP). Typical properties are the provision of multipoint addressing, basic secure channels, and management communications.

- The next reference layer, the *Communication Support Services* module, *CS*, is where the designer should place: basic cryptographic primitives, Byzantine agreement, group communication with several reliability and ordering guarantees, clock synchronization, and other core communication services. The CS module depends on the MN module to access the network.

- Finally, the *Activity Support Services* module, *AS*, should host building blocks that assist execution of intrusion-tolerant applications, such as replication management (e.g., state machine, voting), leader election, transactional management, authorization, key management, and so forth. It depends on the services provided by the CS module.

2.4 Middleware design strategies

The goal of middleware is to support the construction of trusted applications, implemented by collections of components with varying degrees of trustworthiness. This is achieved by relying on distributed fault and InTol mechanisms supplied by that middleware.

Given the variety of possible intrusion-tolerant applications, several different strategies are pursued in order to achieve the above-mentioned goal. These strategies are applied at several levels of abstraction of the architecture, most importantly, in the implementation of the middleware and application services. In this section, we describe these strategies: fail-uncontrolled or arbitrary; fail-controlled with local trusted components; fail-controlled with distributed trusted components.

The conventions used for the figures in the following sections are as follows: grey means untrusted (the darker, the “less trusted”); white means trusted; the presence of a clock symbol means a synchronous environment; a crossed out clock symbol means an asynchronous environment; a warped clock symbol means a partially-synchronous environment; a key means a secure environment; dashed arrows means communication that can be interfered with; continuous arrows denote trusted paths of communication.

2.4.1 Arbitrary failure

The fail-uncontrolled or arbitrary failure strategy is based on the no-assumptions attitude discussed in Section 2.2. When a very large coverage is sought, we resort to making no assumptions about time, following an asynchronous model, and we make essentially no assumptions about the faulty behaviour of either the components or the environment. Of course, for the system as a whole to provide a useful service, it is necessary that at least some of the components are correct. This approach is essentially parametric: it will remain correct if a sufficient number of correct participants exist, for any hypothesized number of faulty participants f.

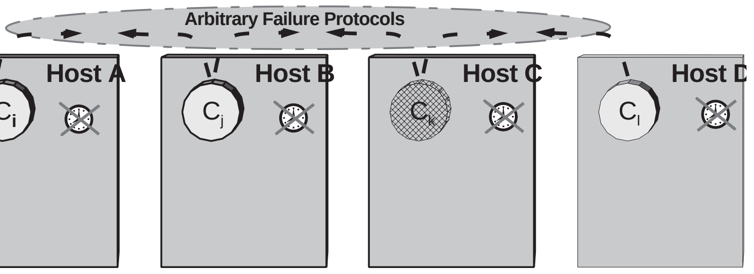


Figure 4: Arbitrary failure

Figure 4 shows the principle in simple terms. The hosts and the communication environment are not trusted, and are fully asynchronous. For a protocol to be able to provide correct service, it must cope with arbitrary failures of components and the environment. For example, component C_k is malicious, but this may be because the component itself or host C have been tampered with, or because an intruder in the communication system simulates that behaviour.

Several protocols in the literature follow this strategy, in order to be resilient to arbitrary failure assumptions. They are Byzantine fault resilient, and have probabilistic or deterministic structure, depending on whether or nor they are subject to the FLP impossibility result [44]. They require a number of hosts $n > 3f$, for f faulty components. Different qualities of service exist, such as: basic binary Byzantine agreement; reliable broadcast; atomic broadcast; multi-valued Byzantine agreement; quorums; state machine replication; etc. This kind of protocols are detailed in Section 3.

Some of these protocols have been featured in the literature under partially synchronous models (models where asynchrony is not absolute) but such a strategy has the problem of introducing vulnerabilities that can be attacked, and as such is not recommended. When this is desired, time should be encapsulated in trusted components under a hybrid model. This problem is further discussed in Section 4.

2.4.2 Fail-controlled with local trusted components

Figure 5 exemplifies a fail-controlled strategy. It consists of assuming that, as for the fail-uncontrolled strategy, hosts and communication environment are not trusted, and asynchronous. However, hosts have a local trusted component (LTC), which supports functions they can trust for certain steps of their operation. This strategy can be substantiated by a hybrid, or wormholes model, with local wormholes, as discussed in Section 2.2. As such, we can construct protocols that cope with a hybrid of arbitrary and fail-silent behaviours, depending on whether a component is interacting with the other components or with the local trusted component (LTC).

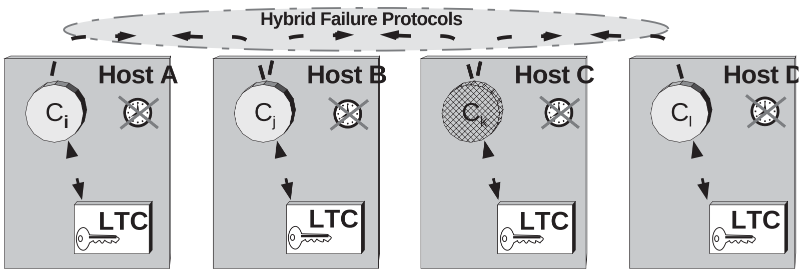


Figure 5: Fail-controlled with local trusted components

In the example, component C_k may be arbitrarily malicious, either because the component itself or host C has been tampered with, or because an intruder in the communication system simulates that behaviour. However, unlike the fail-uncontrolled strategy, the impact of this behaviour on the other components (i.e., error propagation) may be limited, if the protocol makes components perform certain checks and validations with the LTC (for example, signature validation), which will prevent C_k from causing certain failures in the value domain (for example, forging). Likewise, if host B is contaminated, component C_j may behave erroneously, but protocols can be designed in a way that prevents C_j from behaving in an arbitrary (e.g. Byzantine) way towards the other hosts.

2.4.3 Fail-controlled with distributed trusted components

The "fail-controlled with distributed trusted components" strategy amplifies the scope of trustworthiness of the local component support, by making it distributed. As such, certain global actions can be trusted, despite a generally malicious communication environment. This strategy can be substantiated by a wormholes model, with distributed wormholes, as discussed in Section 2.2.

A distributed trusted component (DTC) has the advantage that trust can easily be built on global time-related and security-related properties (such as global time, distributed durations, block agreement). For example, timed behaviour can be supported globally in an intrusion-resilient way, as suggested by the warped clocks in Figure 6: the system is assumed to be *partially synchronous*, that is, anywhere in the interval ranging from time-free to fully synchronous, depending on the environment.

Consider the example of Figure 6, where again component C_k or host C may be arbitrarily malicious. Like the "fail-controlled with local trusted components" strategy, the impact of the faulty behaviour of these components may be limited by enforcing certain validations with the local trusted component. However, the fact that the local trusted components are intercon-

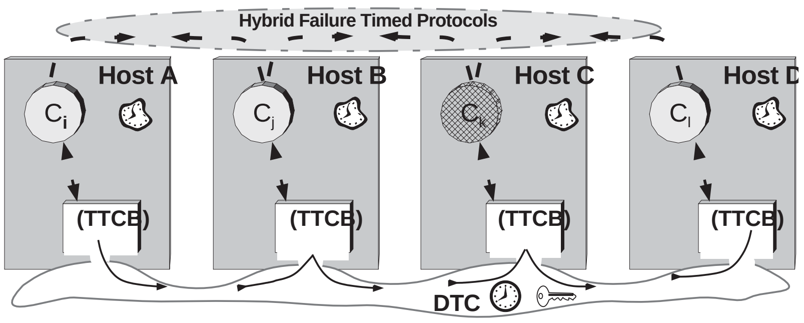


Figure 6: Fail-controlled with distributed trusted components

nected and can exchange information and perform agreement in a secure way — through the control channel — further limits the potential damage of malicious behaviour. For example, a very simple such service might be a generic low-level binary block consensus primitive: the DTC 'knows' directly what each of the payload components in different hosts 'say', unlike the solution with LTCs, where an LTC only 'knows' what a remote component 'says', through information coming through the local payload component. That service could be used, for example, to validate MACs (message authentication codes) used in higher-level protocols [30].

Another advantage of a DTC is the ability to support timed behaviour in an intrusion-resilient way. Timed systems are fragile in that timing assumptions can be manipulated by intruders. The DTC may support trusted time-related services, namely absolute time, duration measurement and timing failure detection, if constructed as a fully synchronous subsystem. The payload system can have any degree of synchronism, as suggested in Figure 6 by the warped clock. The DTC does not make the latter "more synchronous", but allows it to take advantage of its possible synchronism, or to resist attacks against system timing.

Note that all strategies have one thing in common: they assume that both the hosts and communication environment are not trusted and can thus be compromised. Stronger assumptions have been made previously in the literature e.g., that hosts are fortresses or trusted computing bases (TCB). Whilst this may simplify protocol and middleware construction, there is a price to pay in that coverage is lesser and systems become more vulnerable: it is known that the construction of large, complex TCBs (that is, a completely vulnerability free system) is not feasible.

3 Tolerating Intrusions

In the previous section we presented the main InTol concepts and design principles. Here we present the main paradigms for designing intrusion-tolerant systems: threshold cryptography, quorum systems, state machine replication, mechanisms for confidentiality. In the end of the section we present example architectures.

The section summarizes the most common InTol paradigms, which are based on *error masking*. The idea in virtually all techniques considered is to *replicate* a *service* provided to a set of clients $C = \{c_1, c_2, ...\}$ in a set of n servers $U = \{s_1, ..., s_n\}$. The clients and servers (designated by the common word *processes*) run protocols that mask the failure of up to f servers and a possibly infinite number of clients. A process is said to be *correct* if it follows its protocol or algorithm specification during the runtime of the system, and *faulty* otherwise. In InTol we consider that a server may become faulty due to an intentional action of an attacker, so we have to substantiate in some way the assumption that no more than f servers are faulty. This substantiation is done by assuming that there is *diversity* among the servers, i.e., that they are different so they do not have the same vulnerabilities [78, 97].

InTol aims to achieve the three classical security attributes: *availability* (readiness for providing the service), *integrity* (absence of unauthorized state modifications) and *confidentiality* (no unauthorized disclosure of information) [7]. The first two techniques described in this section aim to enforce the availability and integrity of the service: quorum systems and state machine replication. Then in Section 3.4 we survey techniques that not only ensure availability and integrity but also confidentiality.

Distributed system models Physical reality is always very complex, therefore to reason about it in any science – e.g., physics, chemistry or informatics – we need to simplify this reality using *models*. In distributed systems there are several kinds of models. The *fault model* makes assumptions about the types of faults that can happen in the system and its environment. In InTol, the fault model assumed is usually the arbitrary fault model, in which components like the network, servers and clients are assumed to fail *arbitrarily*, i.e., without restrictions on (physically possible) behaviour. *Arbitrary faults* are often called, somewhat loosely, *Byzantine faults*, after the seminal work by Lamport et al. [75]. An important distinguishing factor in intrusion tolerance fault models is the *malicious* factor, to emphasize that the fault distribution is due to an attacker's action and not to accidental facts of nature. To avoid a completely arbitrary behaviour, more complex to handle, several mechanisms are usually employed to exclude by construction some classes of faults. For instance, the communication is usually assumed to be authenticated, messages impossible to forge, and often reliable, since these properties can be obtained using protocols like IPSEC [68] or SSL/TLS [49]. In this section we consider authenticated reliable communication. There are also *architecturally hybrid fault models* in which different fault assumptions are made about different components of the system architecture (see Sections 2.2 and 2.4).

The *time model* is a set of assumptions about the temporal behaviour of the system. Several works on InTol assume the *asynchronous model*, which makes no assumptions about processing and communication times. This model is used because systems that make assumptions about time can be vulnerable to certain attacks against time (e.g., delaying the communication beyond the assumptions). Other systems extend the asynchronous model with failure detectors [23], weak time assumptions [41] or wormholes [32].

3.1 Threshold Cryptography

Threshold cryptography is an expression that denominates a set of algorithms that seem as if they were designed specifically for intrusion tolerance, although they appeared in a quite different context: information security. These algorithms are often important building blocks of intrusion-tolerant paradigms and systems, so we introduce them up-front. There is an extensive bibliography about the area. A nice survey can be found in [52].

There are two basic forms of threshold cryptography. Consider n processes, each holding a secret *share*. The objective of *secret sharing* –first form– is to allow any k of the processes to combine their shares and reconstruct a secret s, ensuring at the same time that $k-1$ (possibly malicious) processes can not do the same, and not even obtain any useful information about s. A *function sharing* algorithm –second form– allows k processes to compute a certain function, while preventing $k-1$ (malicious) processes from doing the same. An especially useful form of function sharing are *threshold signature* functions, which allow k processes to do a digital signature while maintaining the private key secret, since it is not held by any of the processes, but collectively by all.

The relation between these forms of cryptography and InTol is immediate. If up to f servers can fail and k is set to $k = f+1$, then only a subset of correct processes can reconstruct a secret (with secret sharing) or compute a function/signature (with function sharing / threshold signature); on the contrary, a collusion of malicious servers can never do it (since $f = k-1$).

A simple intuition about how threshold cryptography works can be given by one of the seminal secret sharing algorithms, due to Shamir [124] (the other was proposed at the same time by Blakley [16]). The algorithm is based on two properties of polynomials. Consider a polynomial of degree d: $p(x) = a_0 + a_1x + a_2x^2 + ... + a_dx^d$. Given up to d points of the curve defined by $p(x)$ it is impossible to discover the polynomial, but given $d+1$ points it is possible to do it using Lagrange interpolation. Therefore if we define a polynomial $p(x)$ of degree $k-1$ with $p(0) = s$ and give one point of the polynomial to each process, k processes can reconstruct the secret but not $k-1$. This is the basic idea behind Shamir's algorithm. The complete algorithm is more complex, though.

Shamir's algorithm has a few limitations. One is that a process can not verify if its share is "good", i.e., if it serves to reconstruct the secret, so *verifiable secret sharing* algorithms were designed. Another limitation is that if a process provides a corrupt share, the secret that is reconstructed is not the real secret. To deal with this problem, *robust secret sharing* algorithms were designed.

To the best of our knowledge, threshold cryptography was rarely used to build intrusion-tolerant systems. An interesting exception, is the use of a threshold signature scheme in the Mastercard/VISA SET system, in which the private key was shared by several independent organizations [48].

3.2 Quorum Systems

Recall that the objective in this section is to design intrusion-tolerant systems by replicating a service in a set of servers U, accessed by a set of clients C. A *quorum system* $\mathcal{Q}$ is a set of subsets of servers –called quorums– such that $\forall Q_1, Q_2 \in \mathcal{Q}, Q_1 \cap Q_2 \neq \emptyset$ [53, 80]. This definition may not seem particularly enlightening, but it gives the most important idea about quorums: they provide a way to reason about subsets of servers.

Quorum systems for InTol are usually dubbed *Byzantine quorum systems (BQS)*, after the initial work by Malkhi and Reiter [80, 83]. In this section we start by discussing how to implement *data storage services* using BQS. These services are characterized as a set of *shared memory registers*, or variables (each can store one value of a certain domain V). Afterwards, we briefly discuss the implementation of other shared memory *objects* using BQS. Shared memory registers or objects are implemented by the servers in U and accessed (written/read) by the clients[1]. In this section, the BQS are used to ensure the *integrity* and *availability* of the shared memory registers/objects.

3.2.1 Registers

Lamport presented a classification of shared memory registers that is still much used to characterize registers implemented with BQS [73]. A first aspect of that classification is the number of readers (i.e., clients that are allowed to read the register) and writers (clients that are allowed to write the register). Here we consider only the most generic case, i.e., *multi-writer/multi-reader registers*.

A second aspect of that classification is the *consistency semantics* that states what happens to the register when accessed concurrently by several clients. An operation (read, write) o_1 *happens before* and operation o_2 iff o_1 finishes before the beginning of o_2. Two operations o_1 and o_2 are *concurrent* if neither o_1 happens before o_2, nor o_2 happens before o_1. The three Lamport's consistency semantics can be defined in the following way for a certain register x [73, 88]: *Safe:* a read operation that is not concurrent with any write operation returns the last value written in x; a read concurrent with one or more writes returns any value; *Regular:* the same as the safe semantics except that a read concurrent with one or more writes returns either the last value written in x, or the value being written by one of the concurrent writes; *Atomic:* the same as the regular semantics except that reads and writes return values as if they were executed in some order[2]. Each semantics in the list above is stronger than the previous one, since it guarantees the same properties plus some others. In general it is harder to implement

[1]In this chapter we consider that communication is done through a network, and model this communication using *message passing*. *Shared memory* is an alternative communication model that appeared in the context of parallel systems. In distributed systems, shared memory registers/objects have to be implemented using message passing protocols, which is what we do in this section.

[2]Notice that this is not simple to enforce since the register is implemented by a set of servers, not by a single one.

registers with stronger than weaker semantics.

In the context of InTol it is still important to differentiate two kinds of registers: those that tolerate Byzantine faults in the clients (or Byzantine clients for short) and those that do not. Furthermore, registers that do tolerate Byzantine clients can put some restrictions on the kinds of attacks those clients can do.

Register with f-dissemination BQS Let us now illustrate the implementation of a register x with BQS using simple algorithms based on those in [80, 81]. Each server $u \in U$ stores the value of the register x_u and a timestamp $t_{x,u}$. All clients and servers know the content of a certain set W_x that contains the identifiers of the clients allowed to write in x (e.g., it can be stored in another register or be coded in the register's name). For simplicity of presentation, the clients are assumed to communicate with the servers using a *quorum remote procedure call* – Q-RPC. A call to Q-RPC(m) sends a request m to a subset of the servers and collects replies from a quorum of servers. This can be implemented in several ways, for instance first sending m to a quorum of servers, then sending m to more servers until a full quorum replies (since malicious servers may not reply). Neither the clients nor the servers communicate between themselves, which is a property common in algorithms with BQS and that favours scalability [82].

There are several classes of BQS. We implement the register x using one of the simplest, *f-dissemination* quorum systems, which is specific for *self-verifiable data*, i.e., data that a malicious server can not modify unnoticed (e.g., the values stored are signed by the client using its private key and the corresponding public key is known by all processes). The BQS used to implement x has to satisfy two properties:

- *Consistency.* $\forall Q_1, Q_2 \in \mathcal{Q}, |Q_1 \cap Q_2| \geq f+1$
- *Availability.* $\forall Q \in \mathcal{Q}, |Q| \leq n-f$

The first is the condition required for the register to satisfy the desired consistency semantics: that the intersection of any two quorums has at least one correct server. The second is the condition for the register to be always available. The properties allow the quorums to be defined in several different ways. A possible instantiation is to consider $f = \lfloor (n-1)/3 \rfloor$ and the quorums to be any subset of the servers in U of size $\lceil \frac{n+f+1}{2} \rceil$. The simplest case is to consider $n = 3f+1$ and quorums to be any subset of servers of size $2f+1$. This simplified form is in fact an alternative way of defining quorum systems, simpler to understand, used in less theoretical works like [88, 86].

Let us now implement a register x with *regular semantics*, considering initially that clients are always correct. The register is implemented using essentially two algorithms: one for writing a value v in x, and another to read the value in x.

The *client-side write algorithm* is the following: (1) do a Q-RPC to request a set of timestamps $\{t_{x,u}\}_{u \in Q_1}$ from a quorum Q_1; (2) choose the highest timestamp t from $\{t_{x,u}\}_{u \in Q_1}$ and do a Q-RPC to send the signed pair $\langle v, t+1 \rangle$ to any quorum Q_2.

The *client-side read algorithm* is: (1) do a Q-RPC to obtain a set of correctly signed pairs value / timestamp $\{\langle v_u, t_u \rangle_{w_u}\}_{u \in Q_1}$ from a quorum Q_1; (2) return the value with the highest timestamp written by a writer in W_x.

The server-side algorithms are not shown since they are trivially deduced from the client-side ones. It is also simple to understand that the register works as expected. In the write operation, even if the quorum Q_1 contains f Byzantine servers, there are still other $n-2f$ correct servers in that quorum (availability property). In the read operation, if the client receives a value "invented" by a set of Byzantine servers, it discards it because those servers will not be able to impersonate a writer in W_x. Also, a client will always get the most recent timestamp since all quorums intersect in $f+1$ servers (consistency property), at most f of which can be Byzantine. In case there are concurrent writes and a read, the value read is clearly the previous one or one of the values being written, so the semantics is regular.

Let us now consider the problem of Byzantine clients. With the previous algorithms, a malicious writer can leave the register in an inconsistent state by writing a pair with the same timestamp but different values in all servers (poisonous write). The solution is to use an *echo protocol* [118] to ensure that all servers that accept the write, accept the same pair $\langle v, t \rangle$. The modified client-side write algorithm is the following: (1) the writer sends the pair to the servers and obtains signed echoes from a quorum; (2) the writer sends the pair and the signatures to the same quorum. As all quorums intersect, it is impossible to the malicious writer to write two different values with the same timestamp in two different quorums.

Other registers and BQS The requirement for self-verifiable (or signed) data of *f-dissemination* quorum systems is a restriction that may be inconvenient in many cases (for instance, distributing public keys is often difficult in large-scale systems). Therefore, Malkhi and Reiter defined a second class of quorum systems that does not have this restriction: *f-masking* quorum systems. This generalization, however, requires a modification to the consistency property since now the intersection of any two quorums must include a majority of correct servers, i.e., at least $2f+1$ servers: $\forall Q_1, Q_2 \in \mathcal{Q}, |Q_1 \cap Q_2| \geq 2f+1$. The availability property remains the same.

Martin, Alvisi and Dahlin studied the problem of implementing registers with BQS using a minimum of servers [88]. They managed to implement atomic registers (the strongest semantics) with *f-masking* quorum systems and $3f+1$ servers. They also managed to implement regular registers with *f-dissemination* and *f-masking* quorum systems and only $2f+1$ servers.

The problem of Byzantine clients has also received some attention on the literature. The poisonous write mentioned above is only one of the possible attacks a malicious writer can attempt. Recently, Liskov and Rodrigues studied the problem of tolerating other attacks such as exhausting the timestamp space or not completing an algorithm [77].

3.2.2 Beyond registers

BQS can be used to implement shared memory objects other than registers. For instance, Malkhi and Reiter presented a mutual exclusion object [81] and a (randomized) consensus object [82].

The interesting question, however, is: what is the power of BQS to implement intrusion-tolerant services? Do they allow the implementation of registers plus some other objects/services, or any intrusion-tolerant service?

The answer to this question depends on the system model considered. Most works on BQS (including all those cited above) consider that the system is strictly *asynchronous*, with no other time assumptions, failure detectors or other oracles. In fact, the asynchrony of systems based on BQS has always been pointed out as one of its positive aspects. Considering that model, is it possible to implement any service? The answer is no. Although BQS allow the implementation of read and write operations, with different semantics, it was recently shown that they do not allow the deterministic implementation of *update* operations, i.e., operations that modify the state of the shared memory object (or service) taking into account the present state of the service [10]. For instance, $x = 1$ is a write operation, while $x = x + 1$ is an update operation (in C/Java syntax). The implementation of update operations requires timing assumptions (or randomization instead of determinism) [10].

Guerraoui and Vukolic have been studying the performance of BQS without this restriction of a strictly asynchronous time model [56]. Removing this restriction, BQS allow the implementation of state machine replication, which allows the implementation of any intrusion-tolerant deterministic service, the subject of the next section.

3.3 State Machine Replication

State machine replication (SMR) is a generic solution for the implementation of fault-tolerant deterministic services [123], including intrusion-tolerant services [119, 21].

The basic idea is very simple. Consider a service, modelled as a state machine S, that we want to make intrusion-tolerant. The state of S is defined by a set of *state variables* and is modified by a set of *commands*. Commands are atomic in the sense that there can be no interference between their executions. Clients send *requests* for the service to execute commands. SMR consists in replicating S in n servers, and force these servers to emulate the (non-replicated) service in such a way that the service still behaves as expected (i.e., satisfies availability and integrity) even if up to f servers are faulty. This emulation is done by enforcing four properties:

- *Initial state.* All correct servers start in the same state.
- *Agreement.* All correct servers execute the same commands.
- *Total order.* All correct servers execute the commands in the same order.

- *Determinism.* The same command executed in the same initial state in two different correct servers, generates the same final state.

The first property is usually simple to enforce. The second and third are enforced by an (intrusion-tolerant) *atomic multicast protocol* (or total order multicast), which delivers the same requests in the same order to all (correct) servers. The fourth property is always problematic due to the need of diversity among the servers, so specific techniques like software wrappers have to be used to enforce it [22].

SMR algorithms can be classified in two types: *primary-based* and *decentralized*. The former operate in a fashion similar to the classical Lamport's Paxos algorithm [74], so they are often called *Byzantine Paxos* [119, 21, 153, 87, 115]. There is a primary server that orders the requests; when the primary is suspected of being faulty, a new primary is chosen. In decentralized algorithms, the order of the messages is defined in a decentralized way, with the assistance of a consensus algorithm of some kind [18, 32]. We give an example of each below.

3.3.1 BFT

A good example of a *primary-based SMR algorithm* is BFT, the algorithm that showed that Byzantine fault tolerance can be fast, to paraphrase the name of one of the papers about it [21]. The algorithm uses several techniques for efficiency, but probably the most useful is to use message authentication codes (MACs) based on a shared secret key, instead of asymmetric cryptography-based signatures. In BFT, messages multicast to all servers take an *authenticator*, i.e., a vector with one MAC calculated with the secret key shared between the sender and each of the recipients. This authenticator allows the recipient to verify if the message is authentic (i.e., is really from the sender) and has not been modified. Servers always discard messages with an invalid authenticator.

The algorithm is complex and can only be summarized in here. For simplicity we consider $n = 3f + 1$ (in the general case, $f = \lfloor (n-1)/3 \rfloor$). The system evolves in *views*, which are numbered sequentially. In each view, one server is the *primary* and the others are the *backups*. There are two cases: *normal operation* and *view change*. Let us start with the normal operation, which is represented in Figure 7.

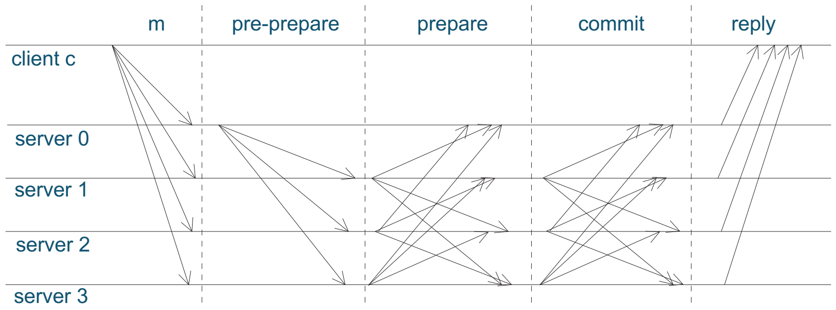


Figure 7: Normal operation of BFT [21].

When a client wants to make a request to the service, it multicasts a message with the command, a timestamp and an authenticator to all the servers (message m in the figure). A client accepts the reply from the service when it receives $f+1$ copies of the same reply from $f+1$ different servers, because this guarantees that at least one of the servers is correct (at most f can be faulty).

When the primary receives a request from the client it runs 3 phases. First, it gives an order number to the message and sends this number with a cryptographic hash of the request to all backups (*pre-prepare phase*). If the backups receive the request from the client and accept the order number (e.g., if it was not given to another request yet and it corresponds to the present view), they send a prepare message (*prepare phase*). Finally, all servers that receive $2f$ prepare messages from other backups, multicast a commit message (*commit phase*). When a server receives $2f+1$ commit messages for a request, the request is accepted for execution. When all requests with lower order numbers are executed, this request is also executed and the result is sent in a reply message to the client.

Proving the correctness of this algorithm is complex and can not be done here, but let us give an intuition that it works as expected. First, a Byzantine primary can not "invent" its own requests, since the backups only process requests that they receive from the client. Second, a Byzantine primary can not give the same order number to two different messages (violating the agreement property) because: (1) a correct backup sends a prepare message only for the first request it receives for a certain order number i; (2) a correct backup sends a commit message only if it receives prepare messages from $2f$ backups; (3) there can not be two different sets of $2f$ backup that send prepare messages for the same i and different requests because $2f+2f+1 > 3f+1$. To conclude: the algorithm gives order numbers to requests; correct servers only execute committed requests; it is not possible to create false requests or give the same number to two different requests.

There is still a problem, however. A Byzantine server can simply decide not to send pre-prepare messages to some requests or to skip some order numbers. To deal with this problem, when a backup receives a request from a client it starts a timer, which is stopped when the request is executed. If the timer expires, the backup informs the other backups that it suspects of the primary. When enough backups suspect of the current primary, a new view is installed.

There are several papers that improve BFT in different ways: terminating faster in "nice" conditions [153, 87], ensuring progress when the primary changes frequently [115], extending it for WANs [5, 4].

3.3.2 SMR with $2f+1$ servers

BFT requires $3f+1$ servers. This number comes from the minimum number of processes needed to solve intrusion-tolerant atomic multicast or consensus, which is equivalent in several system models. Example system models include extensions to the asynchronous model with weak synchrony properties (like BFT), with failure detectors, or with randomization [33].

Although this is a common number of servers in InTol, it is important to reduce it as much as possible, since each server is a machine, with its own operating system, application software and all the management costs involved. In this section we briefly present the only SMR algorithm in the literature that uses less than $3f+1$ replicas, in fact, only $2f+1$ replicas [32]. This algorithm is also an example of a *decentralized SMR algorithm*.

The algorithm was designed under a hybrid distributed system model, where the asynchronous and Byzantine payload part was extended with a component of the *wormholes* class (see Section 2). The wormhole is *not* used to implement the SMR algorithm, but only to provide a simple *ordering service* (just like failure detectors provide a detection service). In this environment, it was possible to achieve an algorithm that reduces the number of replicas to $2f+1$. The system with wormhole-enabled servers is presented in Figure 8.

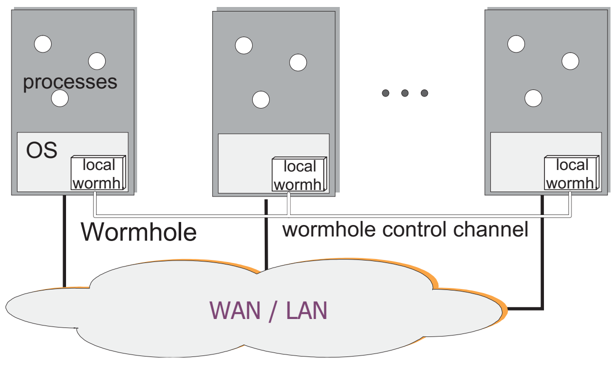


Figure 8: Computers with a *wormhole* [32].

The SMR algorithm works in the following way. A client sends a request protected with an authenticator (vector of MACs) to one of the servers. If after some time it does not get $f+1$ equal replies from different servers, it retransmits the request to other servers. When a server receives the request from the client, it sends the request to all other servers and gives a cryptographic hash of the request to the wormhole. When the other servers receive the request they also give their hashes to the wormhole. When the wormhole collects $f+1$ hashes from different servers, it gives the request an order number and gives that number to all servers. When all requests with lower numbers are executed, this request is also executed and a reply is returned to the client.

Let us give an intuition that the algorithm works as expected. The wormhole is a tiny tamperproof component so it always behaves correctly, giving sequential numbers to the requests. A malicious server can not fool the wormhole into giving order numbers to “invented” requests because the wormhole only gives these numbers when it receives hashes of the request from $f+1$ servers, one of which is correct by assumption, and correct servers only give the wormhole hashes of requests with valid authenticators.

3.3.3 SMR vs. BQS

How does SMR relate to BQS? As seen in the previous section, BQS are a way to reason about quorums of servers, while SMR is a specific form of service replication. SMR is based on a set of servers (U), so we can use BQS to reason about quorums of those servers, but SMR is in fact a more specific technique. A SMR system can be said to be based on BQS if we remove the restriction "quorums $\Rightarrow$ asynchrony", discussed in Section 3.2. The problem is that intrusion-tolerant atomic multicast is equivalent to consensus that can not be solved deterministically in an asynchronous system in which processes (servers in this case) can fail [44]. Therefore, SMR can not be implemented deterministically in asynchronous systems, although it can be implemented using randomized protocols [33] (quite efficiently, as shown by project RITAS [95]).

It is also important to emphasize that SMR is a generic technique that can be used to make any deterministic service intrusion-tolerant. If it was to be used to implement a data storage, then it would provide *atomic* registers, i.e., the strongest of Lamport's semantics. In relation to the implementation of registers with "light" BQS algorithms (like those presented above) versus SMR, some believe that the former are more efficient than the latter, but a recent work shows that this depends on several parameters [37].

A few recent works tried to bring together the best of both worlds, i.e., to use "light" asynchronous BQS algorithms whenever possible, and rely on "heavy" atomic multicast only when needed. The above mentioned work by Bessani et al. uses BQS algorithms for read and write operations and a Byzantine Paxos algorithm only for updates [10]. Cowling et al. presented a SMR algorithm that uses BQS algorithms in normal operation but switches to Byzantine Paxos when it detects concurrency among write requests [36].

3.4 Enforcing Confidentiality

State machine replication and the quorum protocols presented before have the objective of enforcing the *availability* and *integrity* of a service. This section is about ensuring the *confidentiality* of data stored. Recall that the challenge is that not only the clients but also up to f servers can be faulty/malicious and disclose any data stored in them (and/or modify that information and/or deny access to it).

3.4.1 FRS

Although we started with the problem of ensuring availability and integrity of a service, the seminal work that first discussed InTol was actually about a storage service that aimed to ensure the confidentiality of the data stored [47]. This work introduced a technique later called *fragmentation-redundancy-scattering (FRS)* [39]. FRS had the important merit of introducing many of the ideas that reappeared more than a decade later when InTol started gaining momentum.

The basic idea is simple to understand. The service is a file storage, distributed among n servers. Each file F before being stored is fragmented in m fragments, which are scattered among the n servers. Each fragment is encrypted (for confidentiality) and stored in more than one server to guarantee the availability of the file even if there are faulty servers (redundancy). However, no server has enough fragments to reconstruct the file, also to tolerate faulty servers. Detection of corrupted fragments can be done with the assistance of MACs added to each fragment, or by reading several fragments from different servers and voting. The location of the fragments of a file is stored in a specific server, which can also be made intrusion-tolerant using replication.

3.4.2 cAVID

A few years after the appearance of FRS, in a different context, Rabin published a solution to fragment a file – an *information dispersal algorithm* – that optimizes the space used [114]. The idea is to use a *(k,n)-erasure code* to divide the file in n fragments in such a way that it can be reconstructed with k fragments but not with $k-1$. Rabin's scheme was purely mathematical. Krawczyk evolved the scheme to store information in a distributed service [71] and much later Goodson et al. used a similar scheme to implement an intrusion-tolerant file storage with efficient space usage [54].

These works dealt with the problem of efficient storage, but not of confidentiality. That further step was done more recently in [19, 20]. The basic mechanism, which provides only integrity and availability, is called *asynchronous verifiable information dispersal (AVID)*. When a client wants to store/write a file F in the service, it starts by coding F as a vector $[F_1,...,F_N]$ using a *(k,n)-erasure code*, with $k = f+1$ for $n = 3f+1$. Then, it obtains the *fingerprints* of the file [71], i.e., a vector with the hash of each of the file fragments $D = [D_1,...,D_N]$. Finally, it uses a variation of Bracha's reliable multicast protocol [17] to store each fragment F_i in server s_i and the fingerprints in all servers. Each fragment is sent only to its destination server, so the servers may have to reconstruct some of the fragments in case the client is malicious and does not send all fragments. When a client wants to read a file, it simply requests any k fragments and reconstructs the file, using the fingerprints to check if the fragments recovered were modified.

The confidentiality of the file is guaranteed using a variation of AVID, called cAVID. An obvious requirement for confidentiality is than only allowed clients can read the file, so cAVID stores a list L of clients with read access together with the file F. To guarantee the confidentiality of the file, the client starts by generating a random secret key K and then uses it to encrypt the file, which is stored using AVID. The main problem is how to give K to the clients that are allowed to read F. The solution is to use a *(n,k)-threshold encryption scheme*. Each server has a private key SK_i and all clients have the corresponding public key PK. When the client wants to store the file F, besides storing the encrypted file with AVID, it also stores the key K encrypted with PK, and L. When a client c_1 wants to read a file, it sends that request to the servers; if $c_1 \in L$, then the correct servers send c_1 a decrypted share of K and the client uses $k = f + 1$ of those shares to reconstruct the key and decrypt F. On the contrary, no f malicious servers can disclose the content of F because they can not reconstruct K.

Two final notes on InTol confidentiality schemes. Confidentiality is an attribute of data, so these schemes make sense for data storage services. Therefore, the protocols presented can be considered to be quorum protocols. More specifically, cAVID implements a single-writer/multi-reader register. A second note is that *secret sharing* is an obvious candidate to implement confidential storage systems. However, only one system based on this kind of scheme was found in the literature, probably because current secret sharing algorithms are too slow for practical purposes [72].

3.5 Architectures

The previous sections introduced the main paradigms for designing intrusion-tolerant systems: threshold cryptography, quorum systems, state machine replication, mechanisms for confidentiality. In Section 2 we also proposed a reference architecture for intrusion-tolerant middleware. The issue now is how to use these building blocks to build real systems. A few classical paradigms complement the former, like encryption and access control, intrusion detection, firewalls, VPNs, etc. In this section we briefly present actual architectures of intrusion-tolerant systems.

The basic architecture used up to now is composed by n servers accessed by clients through a network. Gupta et al. presented three additional architectures that, however, do not ensure the correctness of the system during all runtime [57]. The first of these architectures is called centralized routing / centralized management. The basic idea is to protect the system using a small set of trusted components. The communication between clients and servers passes though a firewall that does some basic filtering/protection, and a gateway that routes each request to a server. Each request is executed by a single server. If there are intrusions in some of the servers, a component in the servers called configuration management daemon has to detect it and inform a special server, the configuration manager, which restarts the faulty server. In practice it is not possible to ensure that this detection will be made before a malicious server replies to some requests, so this architecture does not mask all intrusions, on the contrary to SMR or registers based on BQS. The other two architectures presented in [57], multicast routing / centralized management and multicast routing / decentralized management, are variations of the same ideas. A similar architecture that deals with dynamic content in the servers was present in [121].

An interesting architecture for SMR was presented by Yin et al. [150]. The architecture is based on the observation that SMR involves two operations – *agreement* on the order of the commands and *execution* of the commands – and that while the first requires $3f + 1$ servers in

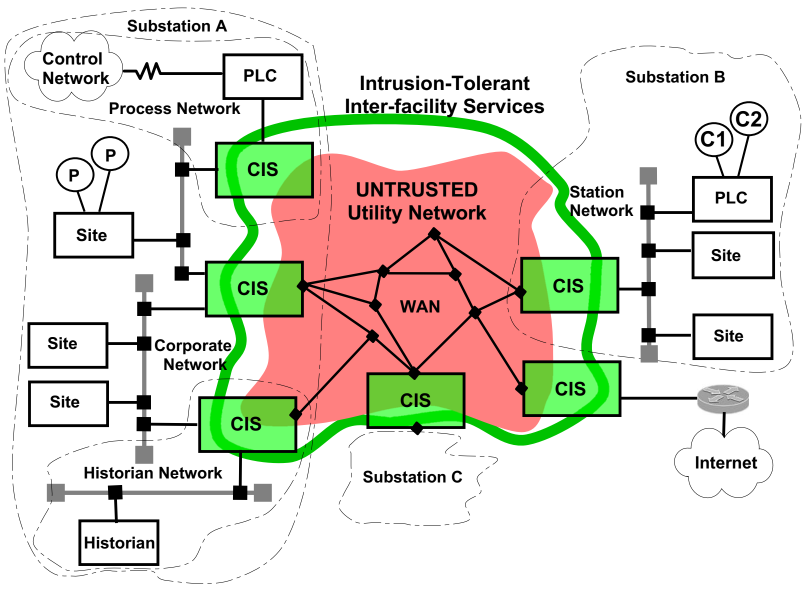


Figure 9: CRUTIAL WAN-of-LANs architecture for critical information infrastructures.

asynchronous systems (plus failure detectors, weak synchrony assumptions or randomization), the latter requires only $2f+1$. Furthermore, in a real system, the execution part, i.e., the service software (say, DBMSs, web servers), is the bulk of the system, while, in comparison, agreement is probably a lighter operation so it can also be done in more cheap machines. Therefore, in this architecture, the clients send the requests to $3f+1$ *agreement servers*, which order them and send them to $2f+1$ *execution servers*, which execute them. The architecture can also include between the two kinds of servers an intrusion-tolerant privacy firewall (with $(f+1)^2$ servers) that guarantees that malicious execution servers do not disclose confidential data.

MAFTIA's precursor architectural work [144] has inspired several intrusion-tolerant architectures, and in fact follows in general terms the reference intrusion-tolerant architecture presented in Section 2.3. It has been used to design several intrusion-tolerant services, such as the SMR service with only $2f+1$ servers of Section 3.3.2 and the Worm-IT group communication system [31]. Trusted support software, some of which taking advantage from a few trusted hardware modules, is wrapped in the run-time environment, extending operating system capabilities and hiding heterogeneity amongst host operating systems. A modular and layered intrusion-tolerant middleware offers the top interface of the MAFTIA architecture to applications, implementing the notion of trusted distributed components. Some native application services run on top of the middleware and exist as default in any MAFTIA system: authorization, intrusion detection, and trusted third party services. Applications built to run on top of MAFTIA are supposed to use the abstractions provided by the middleware and the application services to operate securely across several hosts, and/or be accessed securely by users running on remote nodes, even in the presence of malicious faults.

CRUTIAL is an architecture following the same reference model, proposed for the protection of Critical Information Infrastructures (CII) in general, and power infrastructures in par-

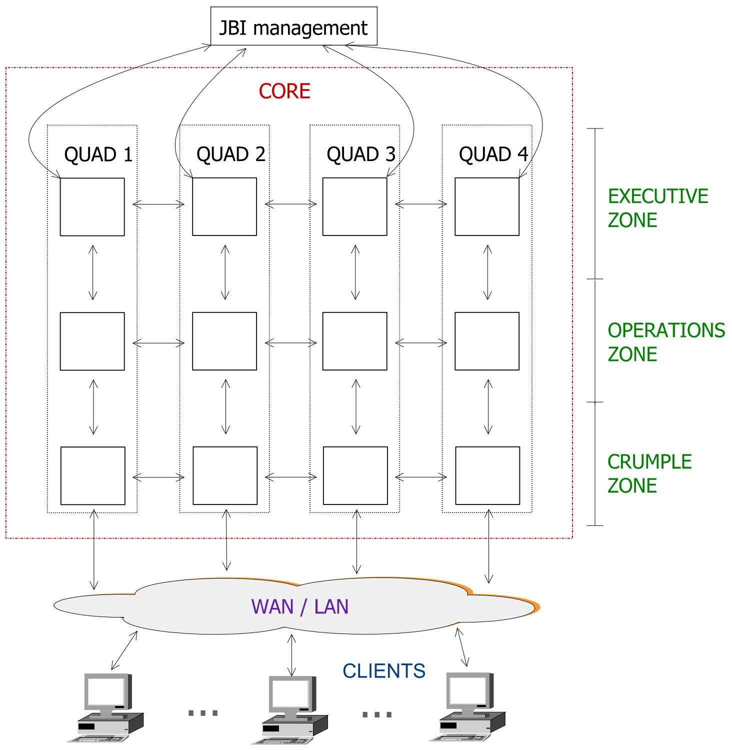


Figure 10: DPASA/JBI architecture [103].

ticular [146]. An infrastructure is modelled as a WAN-of-LANs (see Figure 9). The facilities of the CII are modelled as LANs that are interconnected by a WAN, which is not trusted. The LANs and the WAN are logical entities. For instance, a LAN can be a set of LAN segments, a control network, or even boil down to a single computer. The architecture is hierarchical, so a LAN can also be "zoomed in" and observed as a WAN-of-LANs. LANs can range from trusted to untrusted. An important objective in CRUTIAL is to control/filter the information flow between LANs, so the communication to/from a LAN is protected by a component called a CRUTIAL Information Switch (CIS) [127]. The CIS has to be highly secure due to the criticality of CIIs, so it is made intrusion-tolerant using replication, building on the hybrid/wormhole architecture presented before.

An interesting architecture in the literature is DPASA, an architecture designed, implemented and evaluated in the DARPA OASIS Dem/Val program. The objective was to implement an intrusion-tolerant version of the Joint-Battlespace Infosphere (JBI), a publish-subscribe-query application for military purposes [135, 103]. A simplified presentation of the architecture is in Figure 10. Each square represents typically more than one machine (e.g., two or three).

The core of the system is composed by four quadrants (quad 1 to 4). The quadrants replicate the same functions, so they are equivalent to the servers in the simple architecture we considered above for SMR and BQS. The machines from each of the quads run their own operating system (SELinux, Solaris, Windows XP and Windows 2000). The network adapters of all computers have an embedded firewall that do packet filtering and create VPNs with the machines they are supposed to communicate. The core is also divided in three zones. The crumple zone is the one that holds the first impact from an attack and has the middleware endpoints. The operations zone is where the publish-subscribe-query service itself is executed. The executive zone is used for system management and control, including intrusion detection and correlation. Replication among quads is based on SMR and BQS algorithms.

4 Resisting Attacks

Intrusion-tolerant systems may have a long lifetime (e.g., banking systems, web transaction servers) and the continued production of attacks may lead to the exhaustion of the necessary replicas. In consequence, a sometimes forgotten requirement of practical systems is that it should be guaranteed that no more than the maximum number of tolerated faults ever occurs. In other words, intrusion-tolerant systems should be *exhaustion-safe*.

One effective way of achieving this goal and keeping systems working perpetually is to enhance intrusion-tolerant systems with recovery mechanisms which re-establish the required level of redundancy by repairing failed components. These mechanisms may be either reactive or proactive, as will be seen.

However, an interesting observation is that the exhaustion-safety predicate is not just a mere artefact of an implementation, e.g., of there being enough replicas to sustain the assumed level of threat, or of the recovery mechanisms being, say, fast enough. In fact, recent research has shown that the former property may never be attained unless the right distributed systems model is used.

In this section, we characterize the problem and discuss such a model. Then we explain how proactive and reactive recovery can be implemented under such a (hybrid) model. We explain why reactive recovery should be used as a complementary approach to proactive recovery. Two concrete instantiations for two different intrusion-tolerant application scenarios serve to illustrate the concept and guide the reader: secret sharing and state machine replication. Finally, we discuss different ways of introducing diversity in intrusion-tolerant systems.

4.1 Exhaustion-Safety

In Section 3 we explained the concepts of fault model and time model. The architect of an intrusion-tolerant system is responsible, on the one hand, for choosing the fault and time models under which the system will be designed and later operate, and on the other hand, for ensuring that those models correspond to what happens in the real world, i.e., that the assumptions underlying the models have a good coverage in the environment where the system will execute [111]. Part of those assumptions consist in an abstraction of the actual resources the protocol needs to work correctly. For example: when we assume that network messages are delivered within a known bound, we are in fact assuming that the network will have certain characteristics such as bandwidth and latency; when we assume that a service continues to operate despite faults, we are in fact assuming that there is enough redundancy, such as replicas, to compensate for the effect of those faults.

The violation of these resource assumptions may affect the safety or liveness of the protocols and hence of the system. In this section we explain how system models may be augmented with the notion of the evolution of environmental resources along the timeline of system execution and its consequent impact on system assumptions. We are precisely concerned with the event of 'violation of any of the resource assumptions', which we call *resource exhaustion*, and on the conditions for its avoidance. We start by giving a name to failures caused by resource exhaustion.

Definition 4.1 *An exhaustion-failure is a failure that results from accidental or provoked resource exhaustion.*

Our goal is to prevent exhaustion-failures from happening. Therefore, we define exhaustion-safety in the following manner.

Definition 4.2 *Exhaustion-safety is the ability of a system to ensure that exhaustion-failures do not happen.*

Consequently, an exhaustion-safe system is defined in the following way.

Definition 4.3 *A system is said to be exhaustion-safe if it satisfies the exhaustion-safety property.*

We argue that an intrusion-tolerant system, in order to be resilient, has to satisfy the exhaustion-safety property. In other words, a resilient intrusion-tolerant system must be exhaustion-safe.

4.1.1 The model

In order to formally reason about how exhaustion-safety may be affected by different combinations of timing and fault assumptions, we conceived a model in which exhaustion-safety can be formally defined. This model takes in account the relevant system resources and their evolution with time. For this reason, we called it Resource Exhaustion Model (*REX*, for short).

Our model considers systems (e.g., intrusion-tolerant systems) that have a certain mission. Thus, the execution of this type of systems is composed of various processing steps needed for fulfilling the system mission (e.g., protocol executions). We define three events regarding the system execution: *start*, *termination* and *exhaustion*. Only the start event is mandatory to happen: we cannot talk of a system execution if the system does not start executing. The termination and exhaustion events may or may not happen. More importantly, the causal relation between them is crucial to assess system exhaustion-safety.

We now formally define *REX*.

Definition 4.4 *Let A be a system. An A execution is defined by a triple:*
$\mathscr{A} = \langle A_{t_{start}}, A_{t_{end}}, A_{t_{exhaust}} \rangle$, *where*

- $A_{t_{start}} \in \Re_0^+$ *represents the real time start instant.*

- $A_{t_{end}} \in [A_{t_{start}}, +\infty[$ *represents the real time termination instant.*

- $A_{t_{exhaust}} \in [A_{t_{start}}, +\infty[$ *represents the real time instant when resource exhaustion occurs. If* $A_{t_{exhaust}} \leq A_{t_{end}}$, *system correctness may be corrupted through exhaustion-failures.*

So, under *REX*, a system is defined by a set of triples $\mathscr{A}$, one for each of its executions. Next, we formally define what is an exhaustion-safe system under *REX*.

Definition 4.5 *A system A is exhaustion-safe if and only if* $A_{t_{end}} < A_{t_{exhaust}}, \forall \mathscr{A}$.

Definition 4.5 states that a system is exhaustion-safe if and only if resource exhaustion does not occur during any execution. This does not mean that the system fails immediately after resource exhaustion. In fact, a system may even present a correct behaviour between the exhaustion and the termination events. Thus, a non exhaustion-safe system may execute correctly during its entirely lifetime. However, after resource exhaustion there is *no guarantee* that an exhaustion-failure will not happen. Figure 11 illustrates the differences between an execution of an exhaustion-safe and a non exhaustion-safe system. An exhaustion-safe system is always assuredly immune to exhaustion-failures. A non exhaustion-safe system has at least one execution with a period or periods of vulnerability to exhaustion-failures where resources are exhausted and where correctness may be compromised.

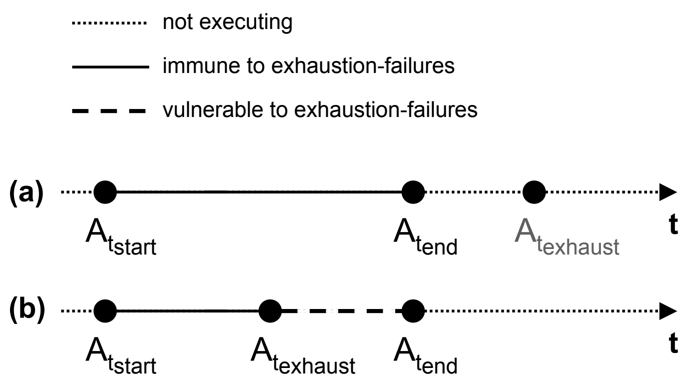


Figure 11: (a) Exhaustion-safe system; (b) non exhaustion-safe system.

4.1.2 Nodes as Resources

As explained in Section 3, intrusion tolerance is typically obtained by replicating a service in a set of nodes. Therefore, in an intrusion-tolerant system, nodes are important resources, so important that one typically makes the assumption that a maximum number f of nodes can fail during its execution, and the system is designed in order to resist up to f node failures. This type of systems can be analyzed under the *REX* model, nodes being the resource considered, and the exhaustion condition being $n_{fail} > f$, where n_{fail} represents the number of nodes that, during an execution, are failed at any time. In other words, this condition states that the system is exhausted when more than f nodes are failed simultaneously.

Notice that in a system in which failed nodes do not recover, this condition is equivalent to state that exhaustion-safety is guaranteed as long as no more than f node failures occur during the system execution. Thus, according to Definition 4.5, a system whose failed nodes do not recover is *node-exhaustion-safe* if and only if every execution terminates before the time needed for $f+1$ node failures to be produced. In order to build a node-exhaustion-safe intrusion-tolerant system, one would like to forecast the maximum number of failures bound to occur during any execution, call it N_{fail}, so that the system is designed to handle $f = N_{fail}$ failures.

In [129, 128] a set of propositions are presented about the possibility/impossibility of exhaustion-safety for categories of algorithms and system fault and time models. The most important result is the *impossibility of building a node-exhaustion-safe intrusion-tolerant system under the asynchronous model and especially in the presence of a malicious adversary*. The intuition is the following: under the asynchronous model it is not possible to upper-bound a system execution time, and thus it is not possible to guarantee that all executions will terminate before $f+1$ node failures being produced and/or provoked.

4.1.3 Proactive recovery

In order to circumvent this impossibility result, one can add some mechanism capable of increasing the time necessary to produce $f+1$ node failures, such that it becomes unbounded and node exhaustion is avoided. One possible way of achieving this is by making use of proactive recovery [102], which can be seen as a form of dynamic redundancy [126]. The aim of this mechanism is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/faults. If rejuvenations are performed sufficiently often, then an adversary is unable to corrupt enough resources to break the system. Proactive recovery has been suggested in several contexts. For instance, it can be used to refresh cryptographic keys in order to prevent the disclosure of too many secrets [60, 59, 51, 152, 18, 151, 85]. It may also be utilized to restore the system code from a secure source to eliminate potential transformations carried out by an adversary [102, 21]. Moreover, it may encompass the substitution of software components to remove vulnerabilities existent in previous versions (e.g., software bugs that could crash the system or errors exploitable by outside attackers). Vulnerability removal can also be done through address obfuscation [13, 12, 45, 106, 149, 112], which could be used to periodically randomize the memory location of code and data objects.

However, the effectiveness of proactive recovery is affected under asynchronous settings: in short, in an asynchronous system a replica can delay its recovery (e.g., by making its local clock slower) for a sufficient amount of time to allow more than f replicas to be attacked. This may happen because: (i) the replica's clock, though correct, is just very (asynchronously) slow; (ii) the replica has been compromised and actively sabotages recovery. A set of practical problems of existing systems that use asynchronous proactive recovery are presented in [132]. The next section describes an effective way of using proactive recovery in order to overcome those problems.

4.2 Proactive Resilience

The main difficulty with proactive recovery is not the concept but the system context in which it is used. The mechanism is useful to periodically rejuvenate components and remove the effects of malicious attacks/failures, as long as it has timeliness guarantees: it must act faster than the estimated speed at which faults develop. In fact, the rest of the system may even be completely asynchronous – only the proactive recovery mechanism needs synchronous execution. Now, this is impossible to realize under an asynchronous system model, so how do we solve this problem?

This type of requirement is correctly addressed under an architecturally hybrid distributed system model, or wormhole model, as presented in Section 2. One particular instantiation to proactive recovery is the Proactive Resilience Model (*PRM*) [128]. Under *PRM*, the hybrid architecture of a system enhanced with proactive recovery has the following characteristics:

- It is composed of the payload (or normal) subsystem, and the proactive recovery (wormhole) subsystem.
- The former is recovered by the latter.
- Each part is designed and built under a different time and fault model.

The payload subsystem executes the "normal" applications. Thus, the payload synchrony and fault model entirely depend on the applications executing in this part of the system. For instance, the payload may operate under an asynchronous and Byzantine environment.

The proactive recovery subsystem executes the proactive recovery protocols that rejuvenate the applications running in the payload part. With regard to the latter, the recovery subsystem wormhole has just the additional strength in synchrony and fault semantics that allow it to fulfil its job, implemented by specific proactive recovery protocols that are application dependent. In the next sections, we exemplify such a proactive recovery wormhole, and illustrate its use through a couple of application scenarios.

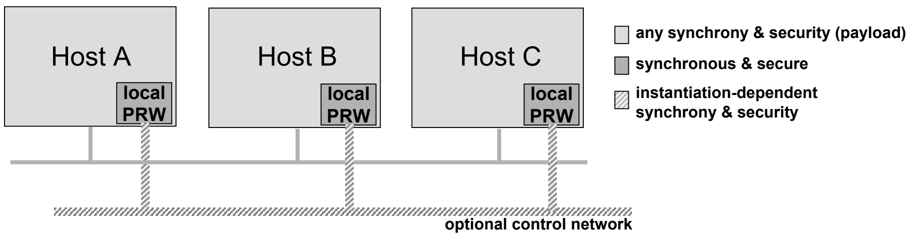


Figure 12: The architecture of a system with a PRW.

4.2.1 Proactive Recovery Wormhole

The architecture of a system with a proactive recovery wormhole (PRW), is suggested in Figure 12: it has a local module in some hosts, called the *local PRW*. Depending on the particular instantiation, these modules may or may not be interconnected by a *control network*. This set up of local PRWs optionally interconnected by the control network is collectively called *the* PRW. This abstract secure and real-time distributed component executes the proactive recovery procedures on behalf of applications running in the payload part of the hosts concerned. This setting can be used in any usual distributed system architecture (e.g., on the Internet).

Conceptually, a local PRW should be considered to be a module inside a host, and separated from the OS. In practice, this conceptual separation between the local PRW and the OS can be achieved in several ways: (1) the local PRW can be implemented in a separate, tamper-proof hardware module (e.g., PC board) and so the separation is physical; (2) the local PRW can be implemented on the native hardware, with a virtual separation and shielding implemented in software, between the former and the OS processes.

The local PRWs are assumed to be fail-silent (they fail by crashing). Every local PRW preserves, by construction, the following property:

P1 There exists a known upper bound $T^{local}_{exec_{max}}$ on the processing delays.

As mentioned, a PRW instantiation may or may not have a control network. For instance, if a proactive recovery procedure only requires local information, then the control network is expendable. Even when the control network is required, its characteristics will depend on the specific requirements of the proactive recovery procedure.

The PRW offers a single service, defined as follows:

Definition 4.6 *Given any function F, with a calculated worst case execution time of T_{Xmax}, an execution interval T_D, and a time interval (period) T_P, satisfying $T_{Xmax} < T_D < T_P$, then F is triggered by the PRW* ***periodic timely execution service*** *at real time instants t_i (the i-th triggering occurs at instant t_i), with $T_D < t_i - t_{i-1} \leq T_P$, and F terminates within T_D from $t_i, \forall i$.*

In short, the PRW has the ability to periodically execute well-defined functions in known bounded time. Moreover, the PRW allows the definition of a set of fail-safe measures to be triggered in certain situations. For instance, these fail-safe measures may shutdown the system if the *periodic timely execution* service fails to satisfy its specification.

A triple $\langle D, \langle F, T_P, T_D \rangle, S \rangle$ defines a PRW instantiation, such that:

- D represents the set of *data* which is proactively recovered in all nodes;
- $\langle F, T_P, T_D \rangle$ represents the *function* F which is periodically triggered with period T_P and timely executed within T_D of each triggering, through the *periodic timely execution* service, in all nodes. F makes operations over the data defined in D;
- S represents the set of (optional) *self-checking* mechanisms, which have the goal of guaranteeing a fail-safe behaviour of all the nodes.

4.2.2 Building Exhaustion-Safe Intrusion-Tolerant Systems

In order to build an exhaustion-safe f intrusion-tolerant system, one has to guarantee that no more than f (accidental or malicious) faults occur during system execution. If the system maximum execution time is known, then one may choose a sufficiently high f — by endowing the system with sufficient nodes — so that exhaustion never occurs. However, if the system has an unbounded execution time, we have a problem — it is not possible to estimate how many nodes will be needed to avoid exhaustion. One possible approach to solve this problem is to use the Proactive Resilience Model — enhance the system with a PRW in order that nodes are periodically and timely rejuvenated. Notice that this approach may even be applied in systems with a known bound on execution time when there is a need for minimizing the number of used nodes.

We propose a design methodology to build exhaustion-safe f intrusion-tolerant systems, under the Proactive Resilience Model. The methodology has 3 steps.

1. Define the data D to rejuvenate, the rejuvenation procedure F, and calculate F's worst case execution time (T_{Xmax}). Then, define the execution interval T_D (greater than T_{Xmax}), and the periodicity T_P (greater than T_D). Finally, define the actions S to be performed if F is not executed with the required periodicity and execution time.
2. Build a PRW instantiation $\langle D, \langle F, T_P, T_D \rangle, S \rangle$.
   - Notice that T_P and T_D may be increased if necessary. This will only impact the required fault tolerance degree, as explained in step 3.
3. Define the degree f_{safe} of fault tolerance, such that, the minimum time necessary ($T_{exhaust_{min}}$) for $f_{safe}+1$ faults to be produced satisfies the condition
   $T_{exhaust_{min}} > T_P + T_D$.

Given that at most f_{safe} faults are produced during any two consecutive rejuvenations, it is guaranteed that no more than f_{safe} faults will ever be produced at the same time during the entire execution of the system.

4.3 Reactive Recovery

As described in the previous sections, proactive recovery is crucial if one wants to build intrusion-tolerant system components that are simultaneously exhaustion-safe. Reactive recovery can be seen as a complementary approach to proactive recovery, in the sense that it may trigger recoveries sooner when malicious behaviour is detected. For instance, one can configure an intrusion detector in order that it triggers a recovery when an intrusion is detected. These early recoveries may have benefits not only in terms of performance, but also in terms of system safety. Proactive recovery guarantees exhaustion-safety as long as recoveries are faster than fault production, i.e., if recoveries take less time than a lower-bound on the time needed to produce $f+1$ node failures. This lower-bound is calculated at deployment time and should be conservative, in order to achieve a very high coverage. However, during system execution, malicious adversaries may prove to be more fierce than expected and may have the ability to compromise $f+1$ nodes within the interval between two consecutive recoveries. Proactive recovery alone is not sufficient to maintain exhaustion-safety in this scenario (because design time assumptions were violated), but reactive recovery has the ability to defend the system against such fierce attacks if it is possible to detect the malicious behaviour of some nodes before $f+1$ being compromised.

4.4 Diversity Management

The different nodes of a distributed intrusion-tolerant system need to be different - or diverse - in order to have a different set of vulnerabilities. Otherwise, an attack that is effective against one node is effective against all of them, and a coordinated attack could compromise all the nodes at almost the same time. Many different diversity techniques have been proposed in the past targeting accidental and/or malicious faults. For instance, design diversity [117] and N-version programming [24] consider only accidental faults. On the other hand, [65] is an early discussion on using diversity to improve security, and more recently [78] presented an important study on diversity in the security domain. [97] identify several possible axes of diversity, i.e., several components of a system that may admit different instances: application software, administrative domain, physical location, operating system, and hardware.

Recoveries should also introduce diversity. This can be done by randomizing certain parts of the system in order that vulnerabilities are somehow changed or removed, and the adversary cannot make use of knowledge learnt before the recovery. Such randomization can be achieved, for instance, through address obfuscation. By using this approach it is possible to randomize the memory location of code and data objects in each recovery. Several techniques have been developed to achieve address obfuscation [45, 106, 12, 149, 13].

Randomization can also be applied to operating system functions [28] and instruction sets [67, 8]. The former allows to mitigate buffer overflows through different types of randomization: system call mappings, global library entry points, and stack placement; while the latter disrupts binary code injection attacks by randomizing their effect.

4.5 Application Scenarios

This section describes two examples of application scenarios where proactive resilience can be applied. Section 4.5.1 describes the design of a distributed f intrusion-tolerant *secret sharing system*, which makes use of a specific instantiation of the PRW – the Proactive Secret Sharing Wormhole – targeting the secret sharing scenario. Section 4.5.2 describes a resilient f intrusion-tolerant *state machine replication architecture*, which guarantees that no more than f faults ever occur while ensuring availability. The architecture makes use of another instantiation of the PRW – the State Machine Proactive Recovery Wormhole – to periodically remove the effects of faults from the replicas.

4.5.1 Proactive Secret-Sharing Wormhole

As explained in Section 3.1, secret sharing schemes protect the confidentiality and integrity of secrets by distributing them over different locations. In many applications, a secret s may be required to be held in a secret-sharing manner by n share-holders for a long time. If at most k share-holders are corrupted throughout the entire lifetime of the secret, any $(k+1,n)$-threshold scheme can be used. In certain environments, however, gradual break-ins into a subset of locations over a long period of time may be feasible for the adversary. If more than k share-holders are corrupted, s may be stolen. An obvious defence is to periodically refresh s, but this is not possible when s corresponds to inherently long-lived information (e.g., cryptographic root and other long-term keys, legal documents).

Thus, what is actually required to protect the secrecy of the information is to be able to periodically renew the shares without changing the secret. Proactive secret sharing (PSS) was introduced in [60] in this context. In PSS, the lifetime of a secret is divided into multiple periods and shares are renewed periodically. In this way, corrupted shares will not accumulate over the entire lifetime of the secret since they are checked and corrected at the end of the period during which they have occurred. A $(k+1,n)$ proactive threshold scheme guarantees that the secret is not disclosed and can be recovered as long as at most k share-holders are corrupted during each period, while every share-holder may be corrupted multiple times in several periods.

The Proactive Secret Sharing Wormhole (PSSW) [131] is an instantiation of the PRW presented in Section 4.2.1. The PSSW targets distributed systems which are based on secret sharing and the goal of the PSSW is to periodically rejuvenate the secret share of each system node. A PSSW allows to timely trigger periodic share rejuvenations *with bounded execution time*.

The PSSW is defined by $\langle D_{PSSW}, F_{PSSW}, S_{PSSW} \rangle$, such that:

- $D_{PSSW} = \{$ `share` $\}$, where `share` is the secret share to be periodically refreshed.
- $F_{PSSW} = \langle$ `refresh`, $T_P, T_D \rangle$, where the `refresh()` function is is based on the share renewal scheme of [60].

- $S_{PSSW} = \{$ *shutdown if share is not periodically and timely refreshed, as specified by* T_P *and* $T_D\}$.

Applying the methodology presented in Section 4.2.1, to build an exhaustion-safe intrusion-tolerant secret sharing system:

1. $D = \{share\}$, $F = refresh()$, $T_D = c_d T_{exec_{max}}$,
   $T_P = c_p T_{exec_{max}}$, and c_d and c_p are constants with $c_p > c_d > 1$, and $S = S_{PSSW}$.
2. Build the PSSW with the parameters defined in step 1.
3. k is chosen in order that $(c_p + c_d)T_{exec_{max}} <$ *'time needed to compromise* $k+1$ *shares'*.

Notice that in the secret sharing scenario, the degree of fault tolerance (f_{safe}) is represented by k.

4.5.2 The State Machine Proactive Recovery Wormhole

In this section, we describe a possible instantiation of a PRW for state machine replication (see Section 3.3) – the State Machine Proactive Recovery Wormhole (SMW) [130]. The goal of the SMW is to periodically rejuvenate replicas such that no more than f SM replicas are ever compromised.

Let the SMW is defined by the triple $\langle D_{SMW}, F_{SMW}, S_{SMW}\rangle$, such that:

- $D_{SMW} = \{$ `OS code`, `SM code`, `SM state` $\}$, where `OS/SM code` is the code of the operating system/state machine and `SM state` is the state of the state machine. These are the three types of data to be periodically refreshed.
- $F_{SMW} = \langle$ `refresh`, $T_p, T_d\rangle$, where the concrete values of T_p and T_d depend on several factors that will be discussed later in the section, and the `refresh` function is presented as Algorithm 1. Each non crashed local SMW $P_i, i \in \{1..n\}$, executes Algorithm 1 at some point of each time period defined by T_p. The precise execution start instant depends on the recovery strategy. More details can be found in [130].
- $S_{SMW} = \{$switch to a fail-safe state and/or alert an administrator, if the state is not periodically and timely rejuvenated, as specified by T_p and $T_d\}$.

A `refresh` function is presented as Algorithm 1. Regarding this algorithm, we assume that the state of both operating system and local state machine is stored in volatile Random-Access Memory (RAM). Moreover, the state of the local state machine is periodically saved to stable storage. Also, we assume that the local state machine is automatically started after every boot of the operating system, and that the previous state is loaded from the stable storage.

In Algorithm 1, line 1 shutdowns the operating system, and consequently stops the execution of the local state machine. Notice that the algorithm continues to execute even after the operating system being shutdown. This happens because the SMW does not depend on the operating system, which can be achieved in practice by implementing each local SMW in a PC board. Line 2 checks if the operating system code is corrupted. To accomplish this task, a digest of the operating system code can be initially stored on some read-only memory, and then assessing if it is correct is only a matter of comparing the digest of the current code with the stored one. In line 3, the operating system code can be restored from a read-only medium, such as a Read-Only Memory (ROM) or a write-protected hard disk (WPHD), where the write protection can be turned on and off by setting a jumper switch (e.g., Fujitsu MAS3184NP). In lines 5–6, the state machine code can be checked and restored using similar methods to the ones we used to check and restore the operating system code. Alternatively, both the operating system and the state machine code can be installed on a read-only medium, thus avoiding the execution of lines 2–5. Line 8 boots the operating system from a clean code and thus brings it to a correct state. The local state machine is also automatically started.

Algorithm 1: refresh() for each local SMW $P_i, i \in \{1...n\}$

```
1: shutdownOS()
2: if OS code is corrupted then {restore operating system code}
3:     restoreOScode()
4: end if
5: if SM code is corrupted then {restore state machine code}
6:     restoreSMcode()
7: end if
8: bootOS() {at this point, the OS and the SM can be safely booted because their code is correct}
9: wait until state recovery is finished
```

Given that the state of the local state machine may have been compromised before the rejuvenation, it may be necessary to transfer a clean state from remote replicas. In line 9, we wait until a potential state recovery is finished. State recovery mechanisms are described in [130]. This same work discusses how to deploy a recovery strategy that guarantees that: (i) no more than f replicas are ever corrupted; and (ii), the execution of the replicated state machine is never interrupted, i.e., the replicated state machine is always available.

4.6 Proactive Resilience in numbers

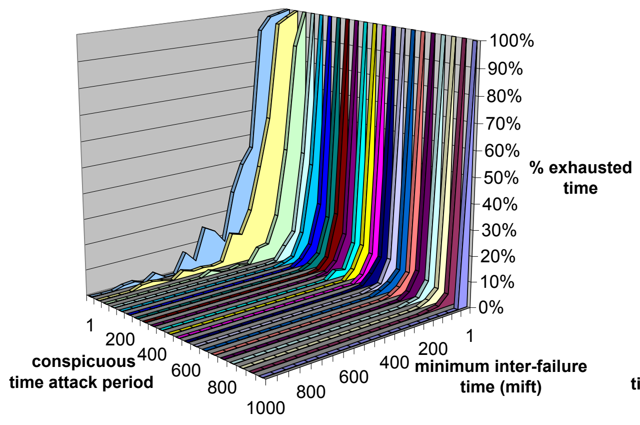
One may ask whether using a more complex (hybrid) model really leads to a more resilient system, in comparison with previous approaches to proactive recovery, namely under the asynchronous model, as reviewed earlier.

An entire set of experiments was made in [133], using the Möbius [38] modelling & simulation tool to assess the advantages of using proactive resilience.

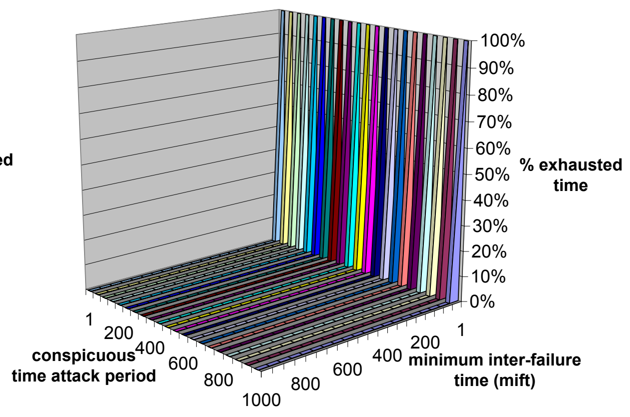
In this section we highlight the main results, focusing on the compared effects of *conspicuous* and *stealth* time attacks, on systems using proactive resilience versus asynchronous proactive recovery. Conspicuous time attacks are attacks that explicitly try to slow down the pace of recovery (e.g., a DoS attack aimed to increase CPU usage and thus increase recovery execution time). Stealth time attacks are attacks that implicitly try to slow down the pace of recovery (e.g., an attack aimed to slow down local clocks and thus increase recoveries interval). Notice that, in theory, stealth time attacks may not even be perceived by the essentially time-free logic of an asynchronous system, leaving it defenceless.

Figure 13 illustrates exhaustion time (measured by the percentage of time in which the system is exhausted) as a function of the combined strength of a conspicuous time adversary and a penetration adversary, for asynchronous recovery (a) and proactive resilience (b). The time adversary periodically slows down the recovery of a different replica, whereas the penetration adversary periodically (period=$mift$) corrupts a different replica.

With asynchronous recovery, the percentage of exhausted time depends on how small the conspicuous time attack period is (Figure 13a), whereas proactive resilience renders the system immune to conspicuous timing attacks (Figure 13b) and the exhaustion time is only affected by the speed of the penetration adversary. Note that, in the latter case, recoveries are triggered and executed by a shielded recovery wormhole that is, by construction, immune to timing attacks launched on the payload system.



(a)

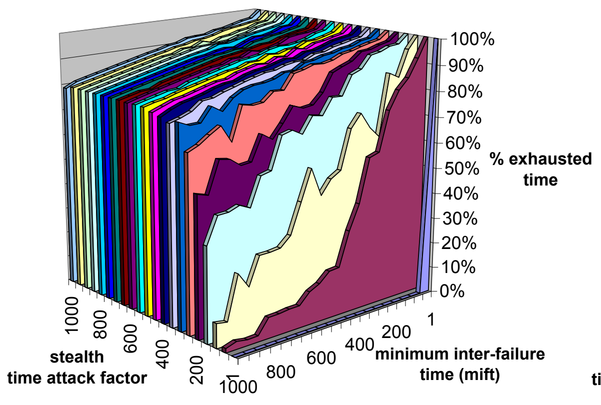


(b)

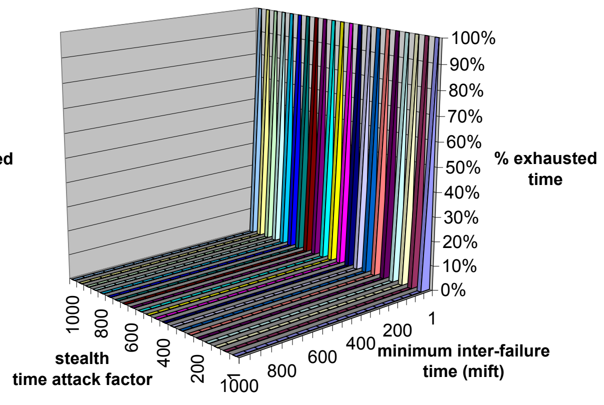
Figure 13: Percentage exhausted time with conspicuous time attacks. (a) Asynch recovery, (b) Proactive resilience.

Figure 14 illustrates, again for asynchronous recovery (a) and proactive resilience (b), a second experiment: the impact of a stealth time adversary that periodically (the period is fixed) slows down the internal clock of a different node. The graphs depict increasing amounts of speed-down (time attack factor). As in the previous scenario, the system exhausts much faster when the time attack factor increases (Figure 14a), whereas proactive resilience also renders the system immune to this second type of attacks (Figure 14b).

The stealth timing attack threat Stealth timing attacks (on, e.g., clocks, timers, or interrupt routines) have not deserved so far a great deal of attention. However, note that these attacks



(a)



(b)

Figure 14: Percentage exhausted time with stealth time attacks. (a) Asynch recovery, (b) Proactive resilience.

are very efficient, since with little power vs. conspicuous direct attacks (e.g., of the denial-of-service (DoS) type) they achieve a more dire effect, as shown in Figure 14a: for attack factors of 200 and up, the system becomes almost permanently exhausted; for an attack factor of 1000, the system is exhausted 80% of the time.

Unlike the conspicuous time attack, in which the delay imposed is proportional to the power exerted, in the stealth attack the amount of delay inserted is virtually independent of the initial power used to gain control of the back-doors to the timing devices.

Some detection is possible in the case of conspicuous timing attacks: the delays injected by the conspicuous adversary may be detected programmatically if the system is partially synchronous and if the internal timebase is not compromised. Moreover, typically these delays affect a large part or the entire system (e.g., DoS) and may get the attention of monitoring devices.

The stealth attacker can more easily evade detection than the conspicuous one. For example, attacks on the internal timebase hit the time references of the system and thus programmatic detections are not reliable, because they use these same time references. Moreover, the attack will not necessarily affect the entire system. For instance, the adversary may tamper with the kernel function that returns the current value of the local clock, and make it return different values to different applications, all the rest working perfectly. Or, alternatively, the attack may be applied to kernel scheduler/dispatcher code, selectively lengthening the execution of functions used by some processes.

Therefore, a stealth time adversary may be very difficult to defend against in classical asynchronous or even partially synchronous systems. The neutralization of this kind of attacks is one of the main achievements of proactive resilience.

5 Testing Attacks and Vulnerabilities

Intrusion-tolerant system designers need to have representative and realistic information about the threats and vulnerabilities their systems will experience once they are put in operation. Otherwise, the implementation of the systems will be flawed because it is based on wrong assumptions (or assumptions with small coverage), and as consequence they will not perform as expected. It is therefore important to understand how intruders attack a system and how they proceed after a compromise, in order to develop dependable and secure systems that are able to cope with a variety of threats.

5.1 Trusting your assumptions

Validating assumptions is often considered not a task of the algorithm or system designer, who considers his/her task finished when, e.g., a successful proof is made that a protocol or algorithm is correct given a set of environmental assumptions. However, let us look at the big picture. Assume one wishes to design an algorithm offering a set of properties A, on a run-time support environment offering a set of properties H. The designer depends on the environment's properties H to implement the algorithm securing properties A. In other words, the designer *assumes* or *trusts* that the environment *has* properties H. What if it does not?

We have just discussed the trust side, now let us observe the trustworthiness side. H holds with a probability Pr_e, the environmental assumption coverage [111]:

$Pr_e = Pr(H|f)$, f - any fault

Pr_e measures the trustworthiness of the environment (to secure properties H). Given H, A has a certain probability of being fulfilled, the coverage Pr_o or operational assumption coverage. It measures the confidence on the algorithm securing properties A (given H as environment), or its trustworthiness:

$Pr_o = Pr(A|H)$

Pr_o can be 1 if the algorithm is deterministic and correct. This is the point of many designers. However, it is easy to understand that:

$Pr_a = Pr_o \times Pr_e = Pr(A|H) \times Pr(H|f) = Pr(A|f)$, f - any fault

As such, in the end of the day, a very elegant and correct algorithm may very well do a poor job, if running over unrealistic or inadequate assumptions.

Particularly important for intrusion-tolerant systems are fault assumptions, and among other things, one would like to know what kind of faults will occur, at what rate they are produced, and if they happen simultaneously or are spread through time. Additionally, designers would like to have a good estimate on the level of vulnerability their systems have after development. This type of information is of utmost importance to keep security risk within certain limits. For instance, given a threat level, one would need to deploy enough replicas with a given degree of vulnerability, to maintain the risk below the desired magnitude.

5.2 Attacker Profile

Since attackers conceal their activities and keep the operational tactics in close secrecy, it is usually very difficult to obtain information about their intents and behaviours. Additionally, attackers can come from anywhere in the world, have different backgrounds and motivations, and as humans, they constantly adapt to evolutions in the surrounding social and technological environment. For many years their malicious activities were largely ignored by the security agencies, which meant that no serious effort was made to systematically collect and divulge data about them. Therefore, even with the important progresses made recently, it is quite challenging to build a profile that reasonably characterizes the attackers.

One of the early attempts to quantitatively describe intrusion actions used computer engineering students to act as hackers [64, 99, 100]. The experiment was conducted over a 4 week period, during which attacks had to be carried out against the department workstations in order to fulfil the requirements of a course. Two interesting results came out from this research. The knowledge level of an attacker can be divided in three stages: low-skilled attackers, which are in the *learning phase*, are mostly unable to perform intrusions, and they still need to educate themselves (e.g., take computer-related courses) to perform attacks in meaningful ways; attackers in the *standard attack phase* search the Internet and exchange information with other hackers, to obtain available expertise on vulnerabilities and exploits; in the *innovative attack phase*, attackers are able to invent new methods and try to exploit unknown vulnerabilities. During the standard attack phase, it was observed that *time to breach* was exponentially distributed, capturing the idea that activities in this phase are mainly done in a trial-and-error fashion. In another experiment, the attack threat in IRC (Internet Relay Chat) environments was assessed [93]. Here, a combination of artificial (i.e., bots that simulated conversations) and regular users were employed during several weeks. From the recorded data it was noticed that the attack level a user withstands is influenced by the gender of the name he or she uses – for example, users with female names are more likely to receive malicious private messages than users with ambiguous or male names. This implies that in IRC channels most attacks are made by humans selecting targets, rather than automated scripts. More recently, a few other studies were made on the attackers behaviour, which were based on information collected in honeypots.

5.2.1 Honeypots

Honeypots are computational systems that have been used in the security context with a few objectives [134, 61, 108]. They have been employed as decoys, to get the attention of the attackers away from the real systems and to waste their time and efforts. As a consequence, vital resources are kept unharmed until other protective measures can be put in place. Another major application of honeypots has been to study intrusions. In this case, honeypots can contain a number of vulnerabilities and can be successfully attacked, but they maintain detailed information about the various malicious actions. The analysis of this data allows for example a better understanding of the procedures that are followed after a machine compromise, and the capture of hacker tools. Moreover, trends that emerge from the data can contribute to answer such questions as: Where do the attackers come from? What are their goals? Do they always behave in the same way? Are they alone or operating in groups?

The concept of honeypot appeared some time ago, in the context of monitoring and tracking activities done at the Lawrence Berkeley Laboratory [136, 137]. During a reasonable period of time, intruder's actions were analyzed to determine which weakness were being exploited, what were the targets of the attacks, and eventually to locate the person responsible for the break in. Cheswick and Bellovin also describe how to implement and deploy a dedicated honeypot, and give a discussion about ethical concerns associated with their use [27, 9]. At the end of the 90's and beginning of the new millennium, there was an explosion on the number of projects developing systems (hardware architectures and software) whose only purpose was to act as honeypots (see [63] for a list of available solutions). These systems can be implemented on a single machine, or they can be built as a network of several components, which are sometimes called honeynets (for a comparison see [107]). As an example, a sophisticated honeypot can contain various target machines, a firewall to control network traffic, an intrusion detection system, and a logging computer.

A honeypot facilitates the execution of two main tasks: it captures and stores as much data as possible; and it prevents honeypot resources from being utilized to attack other external systems. The main challenge is to perform these actions without being detected by the intruders, and at the same time to give them as much flexibility as possible to do whatever they want within the honeypot. Normally, if less restrictions are placed on the attacker behaviour, more interesting information can be captured. However, it becomes much more difficult to contain the attacker. Related to this idea, honeypots have been classified based on the level of the interaction that they allow [134]. *Low-interaction* honeypots only emulate a few services, but they do not have any real operating system where an intruder can operate on. Improved and more complete implementations of fake services are offered by the *mid-interaction* honeypots. A real operating system, where attackers can upload and install tools, is only provided by the *high-interaction* honeypots.

Data collection in honeypots has lead to the unveiling of many intruder practices. These results have been reported in papers describing for example various kinds of attack methods, honeypot forensics techniques, and trends (see [62] for papers on this topic). Some of the results related to the creation of an attacker profile are summarized next. Most of the studies demonstrate that any machine connected to the Internet can suffer several port scans per hour, on a limited set of ports, which correspond to some of the most used services [109, 110, 105]. In a majority of cases, these probes seem to select targets in a random way, without being influenced by the OS of the machine. Most scan sources do not return to the same target for more than one day (this could be caused by the use of temporary addresses). Port scans per se, however, do not provide a good indication that an actual attack will follow, at least from the same source [105]. A better indicator is provided by a port scan plus a vulnerability scan. This could in part be explained by the observation that attackers use in their activities two sets of machines, one to perform the scans with automatic scripts, and another to do the actual manual break in [110, 2]. An initial statistical modelling of the attack process showed that the time intervals between scans/attacks do not follow a Poisson distribution (which is the traditional assumption for failure production in hardware reliability analysis) [66]. A state diagram describing the attacker behaviour after a machine compromise has also been proposed [116].

5.3 Vulnerability Assessment

This section describes several techniques that contribute to estimate how vulnerable a system is. Some of these techniques can be applied in more than one stage of system development, but others are more specialized. They also have distinct requirements in terms of access to the system – for instance, some need to look into the source code to find flaws, while others only have to be able to provide malicious data to the system under test.

5.3.1 Manual, Static, and Dynamic Code Analysis

The main objective of code analysis is to locate deficiencies in programs and reduce the risks for security. It is usually much less expensive to correct these problems if they are found earlier in the software development life cycle, i.e., during the code production phases or while testing.

Manual analysis is one of the oldest methods for the detection of programming flaws. In order to carry out the analysis, the auditor needs to have a good knowledge about the specifications and architecture of the software and about the various coding errors that result in vulnerabilities. To increase efficiency, and facilitate the life of the auditor, the program should be prepared for the review by including for instance relevant comments and a description of the functionality of each procedure. Manual code analysis however is a tiring activity because procedures have to be scrutinized carefully, and therefore it can take a long time.

Static vulnerability analyzers automate some of the tasks of the auditor, as they look for potential flaws in the source code of the applications [26]. Analyzers process the code line by line, and then produce a report telling where vulnerabilities might exist. Next, the programmer only needs to examine the parts of the code for which there were warnings. Depending on the method utilized in the analysis, more or less flaws are found. One of the most straightforward methods only takes into account the lexical rules of the programming language, and looks for dangerous patterns that are usually associated with vulnerabilities [147, 58, 15]. The main problem with this solution is that it produces a large number of false warnings, wasting effort of the person trying to correct the (non-existing) error. Therefore, throughout the years, several other techniques have been proposed and implemented which utilize abstract syntax trees, model checking, theorem provers, and integer range analysis [148, 46, 25, 11, 101]. The addition of annotations to the code has also been proposed as a way to improve the accuracy of the tools [76]. Currently, a few commercial static analysis products are available and are being applied to all sorts of vulnerabilities, including buffer overflows, format strings, and integer overloads.

Dynamic analyzers examine the program behaviour while it is running. Some of these tools require an understanding of the application's functionalities and the development of specific tests. Others are more generic, and they simply monitor the application's execution looking for erroneous actions. For example, some tools change the run-time environment of programs with the objective of thwarting the exploitation of vulnerabilities. The idea here is that removing all bugs from a program is infeasible, which means that it is preferable to contain the damages caused by their exploitation. StackGuard [35] is a simple compiler extension that provides means to detect invalid changes in the stack return address and to prevent those changes from occurring. StackShield [140] also protects frame and function pointers from being changed. Other tools, such as PointGuard [34], add special code to the original program in order to prevent attackers from producing predictable pointer values. These solutions, however, are intrusive because besides requiring access to the source code, they also modify it.

5.3.2 Vulnerability Scanners

Vulnerability scanners are tools whose purpose is the discovery of security weaknesses in computer systems already in production (or about to be put in operation). Consequently, they have to be employed with a certain care because people already working on the system should not be disturbed by the tests. Vulnerabilities can be of many types and flavours, and they can appear in any system component, from hardware to software. Some of the flaws can only be exploited within the system (e.g., a problem in the OS change directory command), requiring local access by the scanner to be able to find them. Others, however, can be exploited remotely by sending malicious packets to the system. Since vulnerabilities of this second class are much simpler to attack, many times they are critical to the overall security, and are the main concern of the intrusion-tolerant systems designers.

Vulnerability scanners that look for remote exposure points are available with different levels of automation, accuracy of the checks, and reporting capabilities. In their most basic form, which are called *port scanners*, they simply provide information about what ports are open and the OS version executing on the remote machine (an example of a well known port scanner is nmap [50]). Although this information is helpful, since it provides an indication of what services are running remotely, there are still too many details missing to determine if a vulnerability really exists. For example, just consider that an ftp service can be implemented by various programs, which go through many releases. Each of these releases has particular problems that one would like to identify in order to select the best solution to protect the system.

The analysis performed by a vulnerability scanner usually progresses in three steps. First, the scanner interacts with the target, by transmitting several well-crafted malicious messages, to obtain information about its execution environment. Besides discovering the type of OS, it is necessary to figure out what are the specific versions of the programs implementing each service that is running. Then, this information is correlated with the data stored in an internal database, to determine if vulnerabilities have previously been found for these services. A scanner to be effective needs to have a detailed and complete database of the most relevant (ideally all) vulnerabilities, which has to be periodically updated as new attacks appear. In the last step, the scanner needs to report its findings. The report can be simply a list of vulnerabilities, or it can give a more friendly diagnose, where vulnerabilities are ordered by their criticality level, and solutions are suggested for each problem.

Throughout the years, several scanners have been built and made available to users. One of the first examples was the Internet Security Scanner (ISS), created by Christopher Klaus in 1992, and distributed through the Usenet newsgroups [69]. ISS could be used to remotely to probe UNIX systems for a group of vulnerabilities. Another tool from approximately the same time was COPS [42], whose objective was to scan and locate vulnerabilities in the local machine. A few years later, another tool called SATAN (Security Administrator Tool for Analyzing Networks) was deployed [43]. SATAN could perform more checks and had a Web-based interface to display results. Currently, there are many other examples of these tools, some of which have been described in the literature and others are commercial products. Most of them are more sophisticated than the earlier examples, for instance, in the way they report the results. Some examples are: Nessus [139], SAINT [122], QualysGuard [113], McAfee Foundstone Enterprise [89], and PatchLink Scanner [79].

5.3.3 Discovering Previously Unknown Vulnerabilities

Fuzzers are testing tools that look for previously unknown vulnerabilities in software applications. They automatically generate invalid data and give it to a target component for processing [98]. Then, the behaviour of the component is observed while the data is consumed, to determine how well that specific input is dealt with. If a failure is seen, this indicates the presence of some flaw that can potentially be exploited by some adversary. Fuzz was one of the first projects to explore these ideas, and it was designed to test UNIX commands (and was later applied to other OSs) [94]. It generates large sequences of random characters which are used as command-line arguments of programs. Many programs failed to process the illegal arguments and crashed. In the recent years, fuzzers have evolved into more intelligent and less random tools, capable of testing different kinds of software components (see for example, [14, 138, 55, 120, 91]).

Robustness testing probes various APIs of an OS to see how effective the latter is at handling erroneous input conditions. In a majority of cases, the testing engines of these tools have an almost complete knowledge of the many functions that compose the target interface, including the data types of the parameters and the expected return values. Consequently, they can produce relatively smart tests and can detect small deviations from the acceptable execution of a function. Most robustness tools have targeted the internal interfaces, such as the kernel API to user level processes. For instance, the Ballista tool was used to assess several OSs that implement the POSIX standard [70]. Similarly, Shelton et al. have made a comparative study of six variants of Windows [125]. Other example studies with these tools include real time microkernels [6] and middleware support systems like CORBA [104, 84]. More recently, this technique has been applied at the OS device driver interface [3, 40, 90]. Since robustness testing has mainly been used with the internal interfaces, which can not be directly exploited by an external adversary, the discovered problems many times do not put security at risk. Nevertheless, their use in the past has shown that they can be very useful to find out problems in the implementation of the interfaces.

Attack injection is a method for vulnerability discovery that automatically generates a large number of malicious interactions (or attacks), which are then transmitted to the target system while monitoring its behaviour [96]. The first tool to implement these ideas was AJECT, and it was specialized to look for flaws in network servers. The tool was composed of two main modules, an injector running in a remote machine and a monitor located in the server's system. A specification of the communication protocol employed by the server (e.g., IMAP, DNS) was used to generate a broad spectrum of valid interactions, which were then maliciously modified accordingly to some predefined algorithms. The resulting attacks were then sent to the target by the injector. The monitor module closely watched the injection process, and traced the server's execution and some basic resource usage (e.g., number of allocated memory pages and the time spent by the CPU). A vulnerability would be detected upon the observation of a *unusual* server behaviour, such as the reception of SIGSEGV signal.

6 Conclusion

The chapter presented a comprehensive overview of the design and validation of intrusion-tolerant – or resilient – middleware and systems. It started by presenting the main design principles. Then, it presented the main intrusion tolerance paradigms and how they can be used to ensure the integrity and confidentiality of a service. The next part of the chapter dealt with the problem of ensuring resilience, i.e., of ensuring perpetual intrusion tolerance, even if attackers periodically compromise components/hosts. The chapter finished with the problem of understanding how attacks are made and how vulnerabilities can be found.

Intrusion tolerance is still mostly an area of research, but we believe that soon it will serve to design commercial systems and be one of the foundations for a more secure information society.

Acknowledgements

We warmly thank our partners in several related projects over the past few years, for many discussions on the topics of the paper. We also thank our colleagues and students at the Navigators group for their collaboration and feedback on this work. This work was mainly supported by the EC, through project IST-FP6-STREP 027513 (CRUTIAL) and NoE IST-4-026764-NOE (ReSIST), by the FCT through the Large-Scale Informatic Systems Laboratory (LaSIGE) and the CMU-Portugal partnership.

References

[1] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Verissimo, M. Waidner, and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21.* January 2002.

[2] E. Alata, V. Nicomette, M. Kaâniche, M. Dacier, and M. Herrb. Lessons learned from the deployment of a high-interaction honeypot. In *Proceedings of the 6th European Dependable Computing Conference*, October 2006.

[3] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 867–876, June 2004.

[4] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Customizable fault tolerance for wide-area replication. In *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, pages 66–80, October 2007.

[5] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, pages 105–114, June 2006.

[6] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, February 2002.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan.-Mar. 2004.

[8] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 281–289, 2003.

[9] S. Bellovin. There be dragons. In *Proceedings of the Third Usenix UNIX Security Symposium*, pages 1–16, September 1992.

[10] A. N. Bessani, M. Correia, J. S. Fraga, and L. C. Lung. Decoupled quorum-based Byzantine-resilient coordination in open distributed systems. In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications*, pages 231–238, July 2007.

[11] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with BLAST. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 2–18. Springer-Verlag, 2005.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[13] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, August 2005.

[14] T. Biege. Radius Fuzzer, September 2005. http://www.suse.de/˜thomas/index.html.

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

[16] G. R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the National Computer Conference*, volume 48 of *AFIPS*, pages 313–317. 1979.

[17] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, August 1984.

[18] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

[19] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, October 2005.

[20] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 115–124, June 2006.

[21] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[22] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269, August 2003.

[23] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[24] L. Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing 1995, Highlights from Twenty-Five Years*. FTCS, 1978.

[25] B. Chess. Improving computer security using extended static checking. In *Proceedings of the Symposium on Security and Privacy*, pages 160–173, May 2002.

[26] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, pages 32–35, Nov/Dec 2004.

[27] B. Cheswick. An evening with Berferd in which a cracker in lured, endured, and studied. In *Proceedings of the Winter USENIX Conference*, pages 163–174, January 1992.

[28] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, December 2002.

[29] M. Correia, L. C. Lung, N. F. Neves, and P. Verissimo. Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 2–11, October 2002.

[30] M. Correia, N. F. Neves, L. C. Lung, and P. Verissimo. Low complexity Byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[31] M. Correia, N. F. Neves, Lau Cheuk Lung, and P. Verissimo. Worm-IT – a wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software*, 80(2):178–197, February 2007.

[32] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183. IEEE Computer Society, October 2004.

[33] M. Correia, N. F. Neves, and P. Verissimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Computer Journal*, 41(1):82–96, January 2006.

[34] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[35] 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.

[36] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of 7th Symposium on Operating Systems Design and Implementations*, pages 177–190, November 2006.

[37] W. S. Dantas, A. N. Bessani, J. Fraga, and M. Correia. Evaluating Byzantine quorum systems. In *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, pages 253–262, October 2007.

[38] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius framework and its implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[39] Y. Deswarte, L. Blain, and J. C. Fabre. Intrusion tolerance in distributed computing systems. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 110–121, May 1991.

[40] J. Durães and H. Madeira. Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. In *Proceedings of the Pacific Rim International Symposium On Dependable Computing*, pages 201–209, December 2002.

[41] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[42] D. Farmer and E. H. Spafford. The COPS security checker system. In *Proceedings of the Summer USENIX Conference*, pages 165–170, June 1990.

[43] D. Farmer and W. Venema. SATAN - Security Administrator Tool for Analyzing Networks, 1995. http://www.porcupine.org/satan/.

[44] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[45] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, May 1997.

[46] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[47] J. S. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218, August 1985.

[48] Y. Frankel and M. Yung. Risk management using theshold RSA cryptosystems. *Usenix ;login: online*, May 1998.

[49] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 protocol. Netscape Communications Corp., November 1996.

[50] Fyodor. Nmap Security Scanner, 2007. http://insecure.org/nmap/.

[51] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, 2000.

[52] P. S. Gemmell. An introduction to threshold cryptography. *Cryptobytes*, 2(3):7–12, 1997.

[53] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

[54] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, June 2004.

[55] A. Greene. SPIKEfile, July 2005. http://labs.idefense.com/labs-software.php?show=14.

[56] R. Guerraoui and M. Vukolic. Refined quorum systems. In *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems*, pages 8–12, 2007.

[57] V. Gupta, V. Lam, H. Ramasamy, W. Sanders, and S. Singh. Dependability and performance evaluation of intrusion-tolerant server architectures. In *Proceedings of the First Latin-American Symposium on Dependable Computing*, pages 81–101, 2003.

[58] E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In *Proceedings of the Symposium on Networked and Distributed System Security*, February 2003.

[59] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 100–110. ACM Press, 1997.

[60] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 339–352. Springer-Verlag, 1995.

[61] Honeynet Project. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley Professional, August 2001.

[62] Honeynet Project. White papers. http://www.honeynet.org/papers/index.html, 2007.

[63] Honeypots.net. Intrusion detection, honeypots and incident handling resources. http://www.honeypots.net/, 2007.

[64] E. Jonsson and T. Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, April 1997.

[65] M. K. Joseph and A. Avizienis. A fault tolerance approach to computer viruses. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–58. IEEE Computer Society, 1988.

[66] M. Kaâniche, E. Alata, V. Nicomette, Y. Deswarte, and M. Dacier. Empirical analysis and statistical modeling of attack processes based on honeypots. In *Proceedings of the Workshop on Empirical Evaluation of Dependability and Security*, June 2006.

[67] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 272–280, 2003.

[68] S. Kent and R. Atkinson. Security architecture for the internet protocol. IETF Request for Comments: RFC 2093, November 1998.

[69] C. Klaus. Internet Security Scanner, 1993. http://www.cert.org/advisories/CA-1993-14.html.

[70] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, September 2000.

[71] H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, pages 207–218, 1993.

[72] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):818–828, September 2003.

[73] L. Lamport. On interprocess communication (part II: Algorithms). *Distributed Computing*, 1:86–101, 1986.

[74] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[75] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[76] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th Usenix Security Symposium*, pages 177–189, August 2001.

[77] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of the 26th International Conference on Distributed Computing Systems*, June 2006.

[78] B. Littlewood and L. Strigini. Redundancy and diversity in security. In P. Samarati, P. Rian, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004, 9th European Symposium on Research Computer Security*, LNCS 3193, pages 423–438. Springer, 2004.

[79] Lumension Security. PatchLink Scan, 2007. http://www.lumension.com/vulnerability-management.jsp.

[80] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium in Theory of Computing*, pages 569–578. ACM Press, May 1997.

[81] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[82] D. Malkhi and M. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, 2000.

[83] D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 249–257, August 1997.

[84] E. Marsden, J.-C. Fabre, and J. Arlat. Dependability of CORBA systems: Service characterization by fault injection. In *Proceedings of the 21st International Symposium on Reliable Distributed Systems*, pages 276–285, June 2002.

[85] M. A. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January–March 2004.

[86] J. P. Martin and L. Alvisi. A framework for dynamic Byzantine storage. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 325–334, June 2004.

[87] J. P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[88] J. P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, volume 2508 of *LNCS*, pages 311–325. Springer-Verlag, October 2002.

[89] McAfee, Inc. McAfee Foundstone Enterprise, 2007. http://www.mcafee.com/us/enterprise/products/vulnerability_managemen

[90] M. Mendonça and N. Neves. Robustness testing of the Windows DDK. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 554–564, June 2007.

[91] M. Mendonça and N. Neves. Fuzzing wi-fi drivers to locate security vulnerabilities. In *Proceedings of the European Dependable Computing Conference*, May 2008.

[92] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*, pages 214–222, July 1987.

[93] R. Meyer and M. Cukier. Assessing the attack threat due to irc channels. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 467–472, June 2006.

[94] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[95] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Randomized intrusion-tolerant asynchronous services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 568–577, June 2006.

[96] N. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. Neves. Using attack injection to discover new vulnerabilities. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2006.

[97] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, Univ. of Lisbon, September 2006.

[98] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, pages 58–62, March/April 2005.

[99] T. Olovsson, E. Jonsson, S. Brocklehurst, and B. Littlewood. Data collection for security fault forecasting: Pilot experiment. Technical Report 167, Dept. of Computer Eng., Chalmers Univ. of Technology, September 1993.

[100] T. Olovsson, E. Jonsson, S. Brocklehurst, and B. Littlewood. Towards operational measures of computer security: Experimentation and modelling. In B. Randell et al., editor, *Predictably Dependable Computing Systems*, pages 555–572. Springer-Verlag, 1995.

[101] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Proceedings of the 13th International Static Analysis Symposium*, pages 405–424, 2006.

[102] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 51–59. ACM Press, 1991.

[103] P. Pal, F. Webber, and R. Schantz. The DPASA survivable JBI–a high-water mark in intrusion-tolerant systems. In *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems*, pages 33–37, 2007.

[104] J. Pan, P. J. Koopman, D. P. Siewiorek, Y. Huang, R. Gruber, and M. L. Jiang. Robustness testing and hardening of CORBA ORB implementations. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 141–150, June 2001.

[105] S. Panjwani, S. Tan, K. Jarrin, and M. Cukier. An experimental evaluation to determine if port scans are precursors to an attack. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 602–611, June 2005.

[106] PaX, 2001. `http://pax.grsecurity.net/`.

[107] F. Pouget and M. Dacier. White paper: Honeypot, honeynet: A comparative survey. Technical Report RR-03-082, Institut Eurecom, September 2003.

[108] F. Pouget, M. Dacier, and H. Debar. White paper: Honeypot, honeynet, honeytoken: Terminological issues. Technical Report RR-03-081, Institut Eurecom, September 2003.

[109] F. Pouget, M. Dacier, and H. Debar. Honeypots, a practical mean to validate malicious fault assumptions. In *Proceedings of the 10th Pacific Rim International Symposium on Dependable Computing*, March 2004.

[110] F. Pouget, M. Dacier, and V. Pham. Understanding threats: a prerequisite to enhance survivability of computing systems. In *Proceedings of the International Infrastructure Survivability Workshop*, December 2004.

[111] D. Powell. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd IEEE International Symposium of Fault-Tolerant Computing*, pages 386–395, July 1992.

[112] R. Pucella and F. B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 230–241, 2006.

[113] Qualys Inc. QualysGuard Enterprise, 2007. http://www.qualys.com.

[114] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[115] H. Ramasamy and C. Cachin. Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 88–102. Springer-Verlag, December 2006.

[116] D. Ramsbrock, R. Berthier, and M. Cukier. Profiling attacker behavior following SSH compromises. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 119–124, June 2007.

[117] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, 1975.

[118] 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.

[119] M. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, volume 938 of *LNCS*, pages 99–110. Springer, 1995.

[120] J. Roning, et al. PROTOS - Security Testing of Protocol Implementations. Computer Engineering Laboratory, University of Oulu, 1999-2003. http://www.ee.oulu.fi/research/ouspg/protos/.

[121] Ayda Saidane, Yves Deswarte, and Vincent Nicomette. An intrusion tolerant architecture for dynamic content internet servers. In *Proceedings of the 1st ACM Workshop on Survivable and Self-Regenerative Systems*, October 2003.

[122] Saint Corp. SAINT Network Vulnerability Assessment Scanner, 2007. http://www.saintcorporation.com.

[123] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[124] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[125] C. Shelton, P. Koopman, and K. D. Vale. Robustness testing of the Microsoft Win32 API. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 261–270, June 2000.

[126] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation (2nd Edition)*. Digital Press, 1992.

[127] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *Proceedings of the 13th IEEE Pacific Rim International Symposium on Dependable Computing*, December 2007.

[128] P. Sousa, N. F. Neves, A. Lopes, and P. Verissimo. On the resilience of intrusion-tolerant distributed systems. DI/FCUL TR 06–14, Dep. of Informatics, Univ. of Lisbon, Sept 2006.

[129] P. Sousa, N. F. Neves, and P. Verissimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*, pages 98–107, June 2005.

[130] P. Sousa, N. F. Neves, and P. Verissimo. Resilient state machine replication. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 305–309, December 2005.

[131] P. Sousa, N. F. Neves, and P. Verissimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pages 686–690, April 2006.

[132] P. Sousa, N. F. Neves, and P. Verissimo. Hidden problems of asynchronous proactive recovery. In *Third Workshop on Hot Topics in System Dependability (HotDep'07)*, June 2007.

[133] P. Sousa, N. F. Neves, P. Verissimo, and W. H. Sanders. Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 71–80, October 2006.

[134] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Professional, September 2002.

[135] F. Stevens, T. Courtney, S. Singh, A. Agbaria, J. F. Meyer, W. H. Sanders, and P. Pal. Model-based validation of an intrusion-tolerant information system. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 184–194, October 2004.

[136] C. Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484–497, May 1988.

[137] C. Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Pocket Books, 1990.

[138] M. Sutton. FileFuzz, November 2006. http://labs.idefense.com/software/fuzzing.php.

[139] Tenable Network Security. Nessus Vulnerability Scanner, 2007. http://www.nessus.org.

[140] Vendicator. Stack Shield : A stack smashing technique protection tool for Linux, January 2001. http://www.angelfire.com/sk/stackshield/.

[141] P. Verissimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *LNCS*, pages 108–113. Springer, 2003.

[142] P. Verissimo. Thou shalt not trust non-trustworthy systems. In *Keynote at the Workshop on Assurance in Distributed Systems and Networks, with the 26th IEEE International Conference on Distributed Computing Systems*, July 2006.

[143] P. Verissimo. Travelling through wormholes: A new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[144] P. Verissimo, N. F. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch. Intrusion-tolerant middleware: The road to automatic security. *IEEE Security and Privacy*, 4(4):54–62, Jul./Aug. 2006.

[145] P. Verissimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 3–36. Springer, 2003.

[146] P. Verissimo, N. F. Neves, and M. Correia. The CRUTIAL reference critical information infrastructure architecture: A blueprint. *International Journal of System of Systems Engineering*, to appear 2008.

[147] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

[148] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, pages 3–17, February 2000.

[149] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*, pages 260–269, October 2003.

[150] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, October 2003.

[151] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.

[152] L. Zhou, F. B. Schneider, and R. Van Renesse. APSS: proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security*, 8(3):259–286, 2005.

[153] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.

Appendices

6.1 Suggested readings

The following journals regularly present contributions in the area of resilience and intrusion tolerance:

- IEEE Transactions on Dependable and Secure Computing
- IEEE Security & Privacy
- ACM Transactions on Computer Systems

6.2 Online resources

We divided online resources in two categories: software packages and research projects.

6.2.1 Software packages

There are not many software packages publicly available that can be used to build intrusion-tolerant distributed systems. We point out the following ones:

BFT The aim of the BFT (Practical Byzantine Fault Tolerance) project was to develop algorithms and implementation techniques able to build practical Byzantine-fault-tolerant systems. BFT was implemented in C/C++ and it is available at http://www.pmg.csail.mit.edu/bft/

CODEX CODEX (COrnell Data EXchange) is a distributed intrusion-tolerant service for storage and dissemination of secrets. At the same time, it comprises a set of general utility packages that provide useful primitives to build intrusion-tolerant systems. CODEX was implemented in C++ and it is available at http://www.umiacs.umd.edu/ mmarsh/CODEX/

JITT The aim of the JITT (Java Intrusion Tolerance Tools) project was to develop a set of tools and libraries for intrusion tolerance using the Java programming language. The objective was to provide a set of fully functional, clear-designed, building blocks to be used by the research community in Byzantine fault- and intrusion-tolerant systems. JITT was implemented in Java and it is available at http://www.navigators.di.fc.ul.pt/software/jitt/

RT-PSS RT-PSS (Real-Time Proactive Secret Sharing) is a library that provides an implementation of the Shamir's secret sharing scheme and Herzberg's proactive secret sharing algorithm. It can be used to build intrusion-tolerant systems that make use of proactive secret sharing. RT-PSS was implemented in C and it is available at http://sourceforge.net/projects/rt-pss/

TTCB The TTCB (Trusted Timely Computing Base) is a distributed embedded component that provides a set of time and security related services to client applications. It can be used as a fundamental building block for the development of intrusion-tolerant and real-time applications. TTCB was implemented in C/C++ and it is available at http://www.navigators.di.fc.ul.pt/software/ttcb/

6.2.2 Research projects

Here we point out a list of research projects and networks of excellence (ongoing and already finished) that study/studied topics related with resilience and intrusion tolerance.

CRUTIAL CRitical UTility InfrastructurAL resilience
http://crutial.cesiricerca.it/

ESFORS European Security Forum for Web Services, Software and Systems
http://www.esfors.org/

MAFTIA Malicious- and Accidental-Fault Tolerance for Internet Applications
http://www.maftia.org

OASIS Organically Assured and Survivable Information Systems
http://www.tolerantsystems.org/oasis.html

ReSIST Resilience for Survivability in IST
http://www.resist-noe.org/

SecureIST ICT Security & Dependability Taskforce
http://www.ist-securist.org/

TCIP Trustworthy Cyber Infrastructure for the Power Grid
http://www.iti.uiuc.edu/tcip/

6.3 Glossary

Attack. A malicious intentional fault attempted at a computing or communication system, with the intent of exploiting one or more of vulnerabilities in the system.

Availability. The readiness of a system or component for providing its service.

BQS. See *Byzantine quorum system*.

Byzantine fault. A malicious fault, i.e., an attack or an intrusion.

Byzantine quorum system. A set of quorums in which some of the processes/servers can be faulty/Byzantine.

Confidentiality. The absence of unauthorized disclosure of information.

Dependability. The measure in which reliance can justifiably be placed on the service delivered by a system.

Error. A fault when activated leads to an error, an erroneous state of the system which can lead to failure.

Failure. A deviation from the service the system is supposed to provide.

Failure assumptions. See *fault model*.

Fault. The hypothesized cause of a failure (see vulnerability, attack, intrusion).

Fault model. Set of failure assumptions about the system.

Integrity. The absence of unauthorized state modifications of a system or component.

InTol. See *intrusion tolerance*.

Intrusion. An intentionally malicious operational fault resulting from a successful attack on a vulnerability.

Intrusion Tolerance. The tolerance paradigm in security: assumes that systems remain to a certain extent vulnerable; assumes that attacks on components or sub-systems can happen and some will be successful; ensures that the overall system nevertheless remains secure and operational, with a quantifiable probability.

Proactive recovery wormhole. A wormhole used to support proactive recovery, i.e., the periodical rejuvenation of the replicas of a system.

PRW. See *proactive recovery wormhole*.

Quorum. A set of processes or servers.

SMR. See *state machine replication*.

State machine replication. A generic solution for the implementation of fault- and intrusion-tolerant deterministic services.

Trust. The accepted dependence of a component or system, on a set of properties of another component or system.

Trustworthiness. See *dependability*.

Vulnerability. A fault in a computing or communication system that can be exploited with malicious intention.

Wormhole. A modelling approach under which a system is a hybrid of a ‘normal’ part that can exhibit weak properties (e.g. asynchrony and Byzantine failure), and a ‘privileged’ part capable of providing a small set of services with stronger properties (e.g., timeliness, security) that are otherwise not available in the rest of the system. The latter part is called a ‘wormhole’.

Index

architectural hybridization, 7
architecture, 6, 7, 9, 10, 26
attack, 4, 42
    attacker profile, 43
attack injection, 48
availability, 15

Byzantine algorithms, 2
Byzantine fault tolerance, 2
Byzantine faults, 15
Byzantine quorum systems, 17, 24

client, 15
code analysis, 45
confidentiality, 15, 24
consistency semantics, 17
    atomic, 17, 24
    regular, 17, 18
    safe, 17
correct process, 15

dependability, 1
diversity, 36
    diversity management, 36

error, 4
exhaustion-safety, 29, 30

failure, 4
    Byzantine failure, 8
failure assumptions, 7, 42
    arbitrary failure assumptions, 7, 8, 12
    architecturally hybrid failure assumptions, 7
    controlled failure assumptions, 7, 12, 13
    hybrid failure assumptions, 7
fault, 4
fault model, 6
    AVI composite fault model, 4
    hybrid fault model, 7
fault tolerance, 1
faulty process, 15
fuzzing, 47

honeypot, 43

integrity, 15
intrusion, 4, 5
intrusion tolerance, 1, 4, 14

middleware, 2, 6, 9
    middleware design strategies, 11
modularity, 9

Paxos, 21
proactive recovery, 32
proactive resilience, 33, 39
    proactive resilience model, 33
process, 15

quorum systems, 17

reactive recovery, 36
register, 17
replication, 14
resilience, 2, 9

secret sharing, 16, 37
security, 1
server, 15
service, 14
state machine replication, 20, 24, 38
    decentralized, 21, 23
    primary-based, 21
system model, 15

threshold cryptography, 16
    secret sharing, 16
    threshold encryption, 25
    threshold signature, 16
time model, 15
trust, 6
    trusted components, 8, 9
trustworthiness, 6

vulnerability, 4, 42
vulnerability assessment, 45
vulnerability scanners, 46

wormholes, 7, 13, 23, 33, 34, 37, 38