

# Public-Key Cryptography and Availability<sup>\*</sup>

Tage Stabell-Kulø and Simone Lupetti

Department of Computer Science, University of Tromsø, Norway  
pesto@pasta.cs.uit.no

**Abstract.** When the safety community designs their systems to also maintain security properties, it is likely that public-key encryption will be among the tools that are applied.

The security guarantees of this technology are based on a particular model of computation. We present the properties of this model that are relevant in the setting of distributed systems. Of particular importance is that the model has no notion of time.

From this it follows that systems that need to be available must exercise the utmost care before applying public-key encryption in any form. We discuss the relation between public-key encryption and timeliness, the tradeoffs that must be made at design time, and how the property of (lack of) availability might very well contaminate other system components.

## 1 Introduction

It is reasonable to expect that the safety community will be forced to deal with an ever-increasing number of security issues (as opposed to only those related to safety) [1]. To solve security problems it is likely that technology from the security community will be applied. Shared-key and public-key encryption, message digests, and digital signatures are all based on advanced mathematical concepts, and these mechanisms are routinely used in security engineering [2].

As is probably the case in other communities, the world-view of the security community is rarely discussed with outsiders. Among insiders this view is more or less taken for granted, and when presenting to outsiders the technology that has been developed there is no room for lengthy depositions on the inner workings of the computational model.

In the world-view of the security community, the most important goal is often, simply said, the prevention of “bad things” happening in and to the system. This is often considered so important that it is permitted to achieve such prevention in ways that also now and then prevent “good things” from happening. After all, it is better to be secure than sorry, and anything less than perfect security is sometimes taken to be equivalent to no security [3]. *Availability* is one of these “good things”, and even though availability certainly is a security property,

---

<sup>\*</sup> This work has been generously supported by the Research Council of Norway by means of the Arctic Bean project (IKT 2010, project number 146986/431) and the Penne project (IKTSoS project number 158569/431).

more often than not, security is only concerned with upholding other security properties such as non-repudiation, integrity and confidentiality.

To shed light on some of the implications of security technology we will use the approach of negative requirements [4]. As our example of a sophisticated security technology that may negatively affect critical system properties, we take public-key encryption [5]. Instead of presenting how public-key encryption can for example be used to turn a channel with integrity into one with secrecy without the exchange of secrets, we focus on what public-key encryption *cannot do*. By demonstrating what properties can not be achieved with public-key encryption, we can say that in systems where these properties are important, public-key encryption should not be used, or at least great care should be taken in the way it is applied.

It is intrinsic to public-key encryption to block in certain situations (and hence become unavailable) [6]. Thus, any system that relies on public-key encryption will be prone to blocking. In essence: The blocking behavior (and thus the whole issue of unavailability) is closely related to certain security properties of public-key encryption. The problem is that it can be impossible in advance (at design time) to determine under which conditions blocking will occur. Furthermore, when public-key encryption is used in the system, it contaminates other components with its blocking behavior. The core of the issue is that if a system for safety reasons can not tolerate blocking, it can not tolerate to be contaminated by the blocking property of public-key encryption.

The outline of the rest of the paper is as follows. In Section 2 we present terminology, and in Section 3 the model of computation used in the security community in sufficient detail to enable us to facilitate a precise discussion. Then, in Section 4 we discuss properties of public-key encryption with focus on what can not be facilitated. Some alternatives are discussed in Section 5. We conclude in Section 6.

## 2 Terminology

Our terminology follows [7]; Please notice that it is slightly different from that traditionally used in engineering.

**Error:** A design flaw. This can be a specification that is outright wrong, or a lack of specification;

**Availability** Readiness for correct service;

**Reliability:** Continuity of correct service;

**Failure:** The nonperformance or inability of the system or a single component to perform its intended function;

Note that a failure is defined as an event, while an error is a static condition. This means that a failure *occurs*, while an error *remains* in that it is part of the system until it is removed (usually by human intervention). A design can have provisions to counter failures, and those that are properly dealt with obviously have no further consequences. Not doing so is an error.

We further define a *system-wide failure* to be one where the system is not only unable to operate, but where an extra-system intervention is required. For example, if the power supply in a machine fails, no program on the stopped machine is able to act and we have a system-wide failure (for this machine).

*Denial of Service* (DOS) is an attack on (parts of) the system to interrupt its normal mode of operation. Such an attack can take place as part of normal operation (flooding a server with requests, for example), or as an extra-system attack (blocking the flow of water to the air condition to ensure that machines shut down due to over heating). The purpose is often of a secondary nature: By interrupting one part of the system, some other parts are also negatively affected. Well designed and engineered components should be able to recover from a DOS attack, but if the purpose is simply to deny service (for a period of time), recovery does not help on the original problem (which is denial of service) [8, 9].

### 3 Models for computer security

The computer security community has two models of computation that are relevant to us: The one used to represent a distributed system, and the one describing encryption. We now discuss them in turn.

#### 3.1 Distributed Systems

The design and implementation of public-key systems rest firmly on the model that the computer security community use to describe distributed systems. It is possible to describe most of the functional aspects of distributed systems using this model. In order to understand the semantics of public-key technology it is prudent to first discuss the model on which it is supposed to rest.

The fundamental abstractions of computer security is that the system consists of a set of *processes*. Processes interact in order to create and keep a *shared state* but also maintain an internal state, of which we can know nothing except by asking them and analyzing the result.

A process can (and many times it does) fail: there are countless ways in which this can happen. A comfortable assumption is that processes are either running, or they are not; this is called *crash fail* [10]. The idea is that a process that has failed remains in a blocked state, but, upon restart, is able to take the actions necessary to restore (its part of) the shared state. If security is important at all, this model is probably too simplistic and processes are instead assumed to potentially fail also by being malicious; in this case we have a so-called Byzantine (or arbitrary) failure [11]. Notice that assuming Byzantine failure also encompasses insiders that for some reason try to harm the system. In fact, this is probably the most common source of Byzantine failures. Evidently wrong (meaningless) data generated by random failures (often called glitches) will in most cases be detected by checksums and other mechanism, and automatically discarded. Apparently legal (meaningful) data forged by an attacker can on the other hand

be accepted as genuine unless provisions have been taken. Byzantine failure describes, for example, the kind of failures that we can expect in presence of Trojan and viruses and in all other cases where authorization policies are circumvented.

Processes run logically separated from each other, and can communicate solely by exchanging messages. Messages travel, independently, on communication links. No interesting assumptions are usually made about these links. In particular, it is not assumed that a message actually manages to traverse the link, that it does so without being corrupted (with or without malice), that messages sent in order retain their order upon arrival, or that no duplication takes place. Notice that not making any of these assumptions is consistent with the failure model of the Internet. In particular, links can fail completely by dropping all messages.

Links that have failed completely may create *partitions*. In general it is impossible to know how many processes reside in a partitioned portion of a distributed system and for how long the partition will last. The (lack of) assumptions about communications implies also that it is impossible to know the difference between a crashed process and a link that has failed completely. For the same reason it is impossible to distinguish between a process that is acting in a Byzantine way and a link that is modifying messages on the fly. Since interaction with a Byzantine process is not fruitful, Byzantine failures can also be viewed as a failure that creates a partition in the system.

The main implication of this is that, in this model, failures always manifest themselves as communication problems. Any communication problems may create (possibly temporary) partitions. This has profound effects on the notion of progress: is the system constructed in such a way that progress is possible when one (or more) nodes are left out (but not known to have stopped or crashed)? What availability means in the specific system and how it is related to partitions can only be answered by looking at the policies of the system (and, thus, can not be answered in general).

For example, if progress of individual users is the goal of the service offered by the system, it would not be wise to design the system in such a way that a partition can stop a user from carrying on with his work [12]. Otherwise, some sort of majority of processes taking responsibility for the global computation (enforcing availability) could be found even if this implies to arbitrarily make decisions about other processes' state. In technical terms this last problem requires the establishment of *consensus* that in turn can not be solved without blocking for an unbounded length of time in the types of systems we consider [13].

The upshot of all of this is that there are two interesting objects to consider in a distributed system, namely links and processes. Furthermore, there are two modes of failure that concerns us, failure by crashing and Byzantine failure. Both types, regardless of whether it happens in a process or in a link, may lead to partitions. Partitions, by definition, are availability problems.

### 3.2 Encryption

We will assume “perfect encryption” [14]. We make this sweeping assumption in order to confine the discussion to other aspects of public-key encryption than the encryption itself. Perfect encryption only makes sense under a certain technological regime. Perfect encryptions encompass the assumption that no fundamental change in the traditional computational model takes place during the lifetime of the system, such as if quantum cryptography becomes available, it is shown that  $P = NP$ , or some other fundamental property on which encryption rests is changed. In this case, our assumption on perfect encryption does not hold and all results must be examined again.

## 4 Public-Key Encryption

In this section we will discuss how and why public-key encryption gives rise to a choice between availability and security.

Security mechanisms are designed under the assumption that system components maintain their integrity. On one hand this lets us design sophisticated run-time mechanisms that both provides security and high availability, on the other hand practice has shown that alteration of the system state is the first goal of potential attacks. Since it is impossible to guarantee system integrity (for all its possible definitions) it is often vital to have a clear view of the system’s *failure model*.

As our running example we will discuss one of the many possible disruptions in a system where public-key encryption is in use: That of a key being compromised. There are many ways a secret key can leak out, ranging from malice to negligence. In any case, if a key has become known (or it is feared that a key has become known) the key must immediately be *revoked*. The revocation demands that any holder of the key should cease using it without delay, and in general we can expect a revocation to signal a serious security incident. Notice that we are not concerned with the continuous revoking of old keys that is part of the normal operation of the system, which in itself introduces a wide range of engineering issues.

We will discuss several aspects of revocation, but in any case, a solution to the issue of revocation is an intrinsic part of the security properties of public key encryption. A solution must be designed for each and every system even though revocation has nothing to do with the cryptographic properties as such.

The following are well-known problems arising from the application of public-key encryption (see for example [6]), but we have cast them in such a way that the tradeoff between security and availability becomes evident.

### 4.1 Who can revoke a key

It must be known in advance who is authorized to revoke a key.

Obviously, a malicious (or erroneously) revocation of some (or all!) of the keys in the system will most likely be a system-wide failure. It is impossible to arrange

things so that this can not happen (if keys can be revoked at all), but one can make it as unlikely as one desires. For example, by means of certificates we can create a *compound principal* such as “Alice **and** Bob **as** Revoke Authority”[15]. When this regime is in place only Alice *and* Bob (in concert) can revoke a key, and neither Alice nor Bob can revoke keys alone. However, revoking a key now requires both Alice and Bob to be available, and this creates a problem of reliability. In concrete terms, from a security point of view there is now a single point of failure in the system: A successful DOS against either Alice or Bob (or both) will paralyze the authority to revoke. In fact, any partition between Alice and Bob will have this effect, regardless of how it comes about.

Because the principal having authority to revoke keys is very powerful, the mechanisms put in place to control it should involve as many participants as possible to guard against malicious attacks, while at the same time as few as possible to ensure that a key can be revoked without delay.

From this we see that designing and implementing a policy for management of authority to revoke keys involves mainly system-specific issues. The design needs to take into consideration the general threat model of the system, the potential costs of not revoking in a timely fashion, the reliability of the network as a whole, the probability of a malicious entity revoking keys, and a host of other issues. Most of these can not be calculated, and estimates must be made on which to base the decision (be means of simulations, for example).

The design of the mechanisms that are to guard the authority to revoke keys is an exercise in the tradeoff between security on one hand and availability on the other.

## 4.2 How to distribute a new key

After a key has been revoked, a new key must be distributed in some pre-determined manner.

Assume that Charlie’s key has been revoked. Until a new key has been disseminated, Charlie is effectively silenced. No one will be able to send him data without violating system security, and data coming from him will be discarded for the same reason. Or, in other words, the part of the system controlled by Charlie is disconnected and so unavailable. The need for security was deemed higher than the need for availability.

One could lump together the authority to create new keys (and certify them) with the authority to revoke keys, but there is no need to do so. In fact, for reasons of security, you probably should not do so. The problem is that on the one hand the message revoking the key should be spread as fast as possible while on the other hand, (parts of) the system might be paralyzed before a new key can be installed. The window can obviously be made to be zero by always issuing the new key together with the certificate that revokes the old one, but this again requires a co-location of the authority that revokes and the one that “restarts” the system.

It is most likely a system-wide failure if the (possibly combined) principal that issues new keys fails by issuing unwarranted keys. As usual, one can make the reliability of this service as high as one deems necessary at the cost of availability.

### 4.3 How to spread the revocation

The notification that a key has been revoked must be spread to all those that potentially hold the key. One can assume that a key will not be revoked unless there is a reasonable strong belief that the key constitutes a security problem. That is, we can assume that the time from which the key becomes known and until all participants has received the message to revoke the key constitutes a window of vulnerability. The problem is that the nature of the task at hand makes it possible for an attacker to make this window of vulnerability as long as he wants. We will examine this issue below.

There are two means of spreading information (a revocation in this case) in a distributed system: Either the information is pushed, or it is pulled [15].

Pushing the information is the simplest solution in that a message is sent to all participants. However, there is no way of knowing that all participants actually receive the message, and if the number of participants is large and their physical distance great, the probability of success of this approach will be rather low. The alternative, to engage in some protocol, is equivalent to creating consensus. Such protocols can be blocking, and are at best probabilistic, where the probability is a function of the characteristics of the physical network (over which processes do not have control). In this state the system is particularly vulnerable to denial of service attacks as security has been breached and the window is open as long as messages are hindered. In other words, pushing is not very secure.

The alternative to pushing is pulling. Each key is augmented with a certificate that requires the one using it to verify that the key is still valid; the details of such an on-line service for verification can be found in [15]. The problem is that in this case the user is blocked if he can not reach the verification service. Again, this service can be made as reliable as one wishes, at the cost of lowering security (the more servers to update in case of a revocation the longer the window of vulnerability).

Another tradeoff is to use a somewhat less reliable but more secure verification service, but issue the verification certificates with a lifetime. But, again, how long this timeout is, will again be a tradeoff between availability and security that needs to be determined in advance.

### 4.4 Recovery from a leaked key

Assume that the principal authorized to revoke a key has decided that based on the available information, a certain key must be revoked. In many cases this only happens after the fact; it becomes known that at some time in the past some event occurred that endangered (the secret part) of a public key. Let us denote the time at which it is decided that the compromise occurred with  $t$ .

The compromise has two implications: Messages encrypted with the public key after time  $t$  can no longer be assumed to be secret, and signatures made with the key after time  $t$  can no longer be assumed to be authentic without scrutinizing of the events leading up to where the signature being made.

If loss of secrecy and/or authenticity is a system-wide failure, a strategy for recovery must be in place. This strategy will determine who has authority to revoke the key, how to spread the revocation, but also how to deal with all messages encrypted with the key since time  $t$ . This recovery procedure can be utterly complicated, and while it is in progress the system might be very vulnerable against DOS attacks, among other things.

To design and implement such a recovery mechanism, while maintaining all other properties of the system, will require a sage tradeoff between security and availability.

#### 4.5 Public-Key Infrastructure (PKI)

We have on purpose avoided the term PKI in our disposition. In the above examples it is evident that *some* means must be found to disseminate keys and certificates, offer verification (and thus revocation) services, to coordinate the activities of the certification authorities, and so on. We believe that whether this is organized under the umbrella of a PKI, or in some other way does not alter any of the arguments we have presented.

#### 4.6 Summary

We see from the examples above that they all reveal the need for a tradeoff between security and availability. Although there are cases where one must surely be selected before the other, this is in general a difficult task. In particular, in most cases it is of prime importance to lower as far as possible the probability of any system-wide failure. The problem is that even though it is obvious that the system as a whole has a certain probability of failure, actually finding it might not be feasible. In particular, due to the complexity of the system, the actual tradeoff that has been done will often not be visible before recovery is necessary.

We believe that the examples we have shown demonstrates that including public-key encryption in a system gives rise to a large set of issues that must be addressed, and that all of them hinge on the probabilities of (a set of) events to occur (or not occur).

In addition to the issues discussed here, public-key encryption introduces also other security properties that need to be considered. For example, the holder of a public key can anonymously send encrypted messages, and the presence of public-key encryption gives rise to the need for authentication. Moreover, it is impossible to know who holds a public key, and thus to know who will verify signatures in the futures and possibly use the signature for a malign purpose.

## 5 Alternatives

In a system without full physical control over the communications links, encryption is the only means available to ensure authentication (and thus authorization) and integrity. There are three technologies that are readily at hand: Symmetric key (e.g., DES), asymmetric key (public key, e.g., RSA), and hashing (digital fingerprint, e.g., SHA). The challenge is to use them in the most convenient manner.

In general we can say that public-key encryption excels in systems where the participants have no prior knowledge of each others. This is seldom the case, electronic commerce with the general public aside. But most of the abstraction it offers can be also obtained using other technologies. As an example, let us demonstrate how to obtain digital signatures using shared keys only.

Assume the three parties Alice, Bob and a trusted Server. Assume furthermore that rather than being trusted to realize a PKI,  $S$  shares a key with  $A$  ( $K_{AS}$ ) and one with  $B$  ( $K_{BS}$ ), and that  $A$  and  $B$  has exchanged a session key ( $K_{AB}$ ) by some means (for example in concert with establishing the Service Level Agreement). The message

$$A \rightarrow B : \{\{M\}_{K_{AS}}, M\}_{K_{AB}}$$

gives Bob all the evidence he needs to hold  $M$  against Alice, with the assistance of  $S$ . This places the same responsibilities on  $S$  as would the combination of implementing a PKI and a CA. Notice that in both the solution for shared-key and public-key encryption  $S$  must be trusted to be willing and able to do “the right thing”. If  $S$  fails to be trustworthy, both technologies fail to provide a solution. The only difference is precisely what this “the right thing” is. Or, in other words: Establishing digital signatures is a matter of establishing and maintaining trust rather than of cryptographic technology.

Another issue for concern is the ability to keep keys secret. In public-key encryption there is also a key-component that must be kept secret, and the engineering challenges are not smaller for this technology than for shared-key encryption; keeping one key secret is not much more complicated than keeping thousand keys secret. Also in this respect the tradeoff is more biased by trust and belief in the ability of participants to uphold local security policies, than of technology.

Taken together we can say that whether the management necessary to support shared-key encryption is a heavier burden to carry than that of public-key encryption is system dependent. Providing universally valid guidelines is probably impossible.

## 6 Conclusions

The security community has an array of powerful technologies to offer systems designers. We have discussed but one: Public-key encryption for secrecy (encrypting with the public key) and authentication (digital signatures).

Failures are inherent in the computational model of distributed systems, and this creates problems. To uphold security properties, public-key encryption must be supported by complex and distributed infrastructure. Taken together, when public-key encryption is used, a tradeoff must be found between availability on one hand, and security on the other. The availability of the system is heavily affected by this decision.

Public-key encryption introduces many lanes that can lead to system-wide failures, and that can lead to blocking (denial of service) in whole or parts of the system. Unfortunately, there does not seem to be any structured manner in which to proceed, as all tradeoffs must be made based on the actual network topology, the properties of the resources that must be protected, and so on. In particular, it seems as if the tradeoffs *must* be made at the time of deployment.

All of this should be contrasted to the use of symmetric keys. The process of exchanging keys can be complex, and shared keys must be protected; just as the secret part of in the key-pair in a public-key system. But with a shared key it is clear with whom you share a key, it is clear who can authenticate you, anonymous receiver and senders are not feasible, and so on.

The lesson to be learned is that although the somewhat troublesome properties of public-key encryption is well known in the security community, this might not be the case in the safety community. From this it should follow that such powerful abstractions as digital signatures should only be applied if they are fully understood. In particular, if blocking in any form is a problem in the target system, authentication and integrity should be achieved by other means than by using public-key encryption.

This does not necessarily mean to avoid public-key technologies in all cases but to carefully examine all the possibilities not letting its recognized power to mask its drawbacks: When valid alternatives are available the choice can not be obvious.

## 7 Acknowledgments

This paper would not have been written had it not been for the IKTSoS workshop arranged by the Norwegian Research Council in March 2005. Feico Dillema, Jon Ølnes, Arne Helme and Dmitri Zagorodnov acted as catalyzators in crystalizing our ideas.

## References

1. Pfizmann, A.: Why safety and security should and will merge. In Heisel, M., Liggesmeyer, P., Wittmann, S., eds.: Proceedings of Computer Safety, Reliability, and Security (SAFECOMP'04). Volume 3219 of Lecture Notes in Computer Science., Potsdam, Germany, Springer (2004) 1–2
2. Anderson, R.J.: Security Engineering. John Wiley & Sons, Inc. (2001)
3. Lampson, B.: Security in the real world. IEEE Computer **37** (2004) 37–46
4. Rushby, J.: Critical system properties: Survey and taxonomy. Reliability Engineering and System Safety **43** (1994) 189–219

5. Nechvatal, J.: Public key cryptography. In Simmons, G.J., ed.: Contemporary cryptology, the science of information integrity. IEEE Press (1992) 177–288
6. Roe, M.: Cryptography and evidence. PhD thesis, Clare College, University of Cambridge, UK (1998)
7. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1** (2004) 11–33
8. Needham, R.M.: Denial of service: an example. Communications of the ACM **37** (1994) 42–46
9. Mirkovic, J., Reiher, P.: A taxonomy of DDoS attack and DDoS defense mechanisms. SIGCOMM Computer Communication Review **34** (2004) 39–53
10. Barborak, M., Dahbura, A., Malek, M.: The consensus problem in fault-tolerant computing. ACM Comput. Surv. **25** (1993) 171–220
11. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4** (1982) 382–401
12. Stabell-Kulø, T., Dillema, F., Fallmyr, T.: The open-end argument for private computing. In Gellersen, H.W., ed.: Proceedings of the ACM First Symposium on Handheld, Ubiquitous Computing. Number 1707 in Lecture Notes in Computer Science, Springer Verlag (1999) 124–136
13. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
14. Blum, M., Goldwasser, S.: An efficient probabilistic public-key encryption scheme which hides all partial information. In: Proceedings of Advances in Cryptology—Crypto’84. Volume 196 of Lecture Notes in Computer Science., Springer verlag (1984)
15. Lamson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: theory and practice. ACM Transactions on Computer Systems **10** (1992) 265–310