Revisiting Liveness Properties in the Context of Secure Systems

Felix C. Gärtner

École Polytechnique Fédérale de Lausanne (EPFL)
Departement de Systèmes de Communications
Laboratoire de Programmation Distribuée
CH-1015 Lausanne, Switzerland
fgaertner@lpdmail.epfl.ch

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
Technical Report 200278
November 4, 2002

Abstract. Distinguishing trace-based system properties into safety properties on the one hand and liveness properties on the other has proven very useful for specifying and validating concurrent and fault-tolerant systems. We study the adequacy of these abstractions, especially the liveness property abstraction, in the context of *secure* systems for two different scenarios: (1) Denial-of-service attacks and (2) brute-force attacks on secret keys. We argue that in both cases the concept of a liveness property needs to be adapted. We show how this can be done and relate the resulting concepts to related work in the areas of concurrency theory and fault-tolerance.

Keywords: Safety, Liveness, Security, Modeling, Denial-of-service

1 Introduction

It was observed in 1977 by Lamport [24] that system properties can informally be classified into two distinct classes: safety properties and liveness properties. Generally speaking, safety properties state that "something bad never happens", i.e., a certain bad condition will never occur in any system configuration. Mutual exclusion and partial correctness are two prominent examples of safety properties. For the former, the bad condition is that two processes are in their critical section at the same time. For the latter, the bad condition describes a termination state where the postcondition does not hold. Safety properties are a well-established concept and a lot of theory and practice has evolved around it.

In contrast to safety properties, liveness properties demand that "something good eventually happens", i.e., a certain desired condition will eventually be true for some system configuration. The most prominent example of a liveness

property is termination. Liveness properties can be regarded as a first-order approximation of real-time properties. The distinction made is merely that between "finite" and "infinite" time. While safety properties are violated in finite time, liveness properties are violated in infinite time. In a later article, Lamport elaborates on the meaningfulness of a system satisfying a liveness property as follows [25]:

> The question of whether a real system satisfies a liveness property is meaningless; it can be answered only by observing the system for an infinite length of time, and real systems don't run forever. Liveness is always an approximation to the property we really care about. We want a program to terminate within 100 years, but proving that it does would require addition of distracting timing assumptions. So, we prove the weaker condition that the program eventually terminates. This doesn't prove that the program will terminate within our lifetimes, but it does demonstrate the absence of infinite loops.

Despite such drastic statements, liveness properties are a widely accepted concept when modeling and analyzing the timing behavior of algorithms.

Safety and liveness have been considered adequate in the area of fault-tolerant systems too. Specifying systems using safety properties directly translates to this area since safety properties make sense without change in the presence of faults. In the context of silent crash faults it was observed that liveness properties must be restricted to those parts of the system which remain alive. For example, the standard specification of the *consensus problem* [6, 33], the basis of distributed transactions and hence a very important problem in fault-tolerant computing, involves a safety property and a liveness property:

– (Safety) If two processes choose a certain value $v \in \{commit, abort\}$, they choose the same value.
– (Liveness) Every process eventually chooses a value.

The safety property perfectly makes sense if processes can crash. But the liveness property needs to be weakened into

– (Liveness) Every process that doesn't crash eventually chooses a value.

to be implementable, yielding the definition of *uniform consensus*. In other fault settings, the specification needs to be adapted in similar ways (see for example the area of *self-stabilization* [16]).

In this paper we turn our attention to the adequacy of liveness properties when studying *secure* systems. We will not attempt to define what security means in general, but rather look at two individual examples of properties which we intuitively regard as security properties and which we would like to formalize, if possible, as liveness properties. The example properties come up in the context of two different forms of attacks:

– Denial-of-service attacks [12]: In a denial-of-service attack a user is prevented from using a remote resource by, for example, flooding the network with

bogus messages. In this setting we ask: What are sensible forms of liveness to specify the progress properties we require of a system in the presence of such attacks?

- Brute-force attacks on cryptographic (public) keys: In these types of attacks, an adversary tries to compromise the secrecy of a cryptographic key by trying and testing every possible solution from the key space. While this usually takes a prohibitively long period of time, some instances of such attacks are feasible. (For example, it was possible to break an instance of the Data Encryption Standard DES in less than a day [30].) But even when abstracting from concrete time instances (as is done in the domain of liveness), a brute force attack is guaranteed to terminate. (Note that while private (symmetric) key cryptography may in some cases be resilient to brute force attacks, public (asymmetric) key cryptography can *always* be attacked in this way since one key is necessarily disclosed.) In this setting we ask: What are sensible concepts in the spirit of liveness properties to model the resilience of an algorithm to brute-force attacks if an adversary can delay the progress of the algorithm for an arbitrary (but finite) amount of time?

In both cases we describe concepts that can help model the properties in question. For the denial-of-service case we end up with a concept which we call *self-controlled liveness properties*. These properties can be seen as being the particular subset of liveness properties which the adversary cannot control. We define this concept formally and relate it to the similar concept of *machine-closure* [25] from concurrency theory. This work aims in the direction of better understanding system properties in the context of denial-of-service attacks, an open issue recently stated by Meadows [29].

In the brute-force case we describe and advocate concepts of other authors [11, 22] which, we think, deserves more attention. The path in this case is to introduce concepts from complexity theory in a way which complements specifications based on safety and liveness properties. Briefly spoken, the additional *efficiency* property mandates that an adversary does not have the computational power to delay the execution "long enough" (in a complexity theoretic sense).

The paper is structured as follows: We first state the formal background of reasoning about systems in the context of fault-tolerance and security in Section 2. Sections 3 and 4 deal with the cases of denial-of-service and brute-force attacks, respectively. Section 5 concludes the paper.

2 Trace-based System Models and Properties

We now briefly recall the concepts of trace-based specifications for reactive systems.

2.1 Transition Systems and Traces

Usually, an interactive system is modeled as a state machine which moves from one state to another by means of actions. Formally this corresponds to the definition of a *labeled transition system*. In the black box view of systems, we wish to

define the behavior of such a system in terms of the states and actions it exhibits at its *visible interface*. In the literature this is termed *observation semantics* and there are many different possibilities of defining observation semantics for concurrent systems. We will use one of the simplest semantics, called *trace semantics*, which amounts to defining an observation simply as a sequence of states and actions which are visible at the system interface. Formally, a *trace* is written

$$s_1 \stackrel{a_1}{\rightarrow} s_2 \stackrel{a_2}{\rightarrow} s_3 \ldots$$

and denotes that starting from state s_1 the system reaches state s_2 by executing action a_1 etc.

Note that trace semantics can also be used to describe the behavior of concurrent systems by defining a state as being a *distributed state* (i.e., a vector of local states) and viewing a trace as the interleaving of the individual local traces of the concurrent processes. For example, if $s_1 \stackrel{a_1}{\rightarrow} s_2 \stackrel{a_2}{\rightarrow} s_3 \ldots$ is the trace of a process p and $s'_1 \stackrel{a'_1}{\rightarrow} s'_2 \stackrel{a'_2}{\rightarrow} s'_3 \ldots$ is the trace of process p', we can model a trace of the concurrent system where p and p' take turns as:

$$(s_1, s'_1) \stackrel{a_1}{\rightarrow} (s_2, s'_1) \stackrel{a'_1}{\rightarrow} (s_2, s'_2) \stackrel{a_2}{\rightarrow} (s_3, s'_2) \stackrel{a'_2}{\rightarrow} (s_3, s'_3) \ldots$$

The type of interleaving can be used to distinguish different synchrony assumptions between processes. One of the most general assumptions is that in an infinite trace both processes must take steps infinitely often. Since this rules out one process dominating the trace, this concept is often called *fair interleaving*.

2.2 Safety and Liveness

A *property* is defined to be a set of traces. A trace σ *satisfies* a property P if $\sigma \in P$. If σ does not satisfy P we say that σ *violates* P. There are two important types of properties called *safety* and *liveness* [2]. Informally spoken, a safety property demands that "something bad never happens" [24], i.e., it rules out a set of unwanted trace prefixes. Mutual exclusion and partial correctness are two prominent examples of safety properties. A liveness property on the other hand demands that "something good will eventually happen" [24] and can be used to formalize, e.g., notions of termination. Safety and liveness properties are defined as follows.

A *safety property* is a property S such that for each trace σ that violates S, there exists a finite prefix α of σ such that for all traces β, $\alpha \cdot \beta$ violates S (the dot "$\cdot$" denotes concatenation). A *liveness property* is a property L such that for all finite traces α exists a trace β such that $\alpha \cdot \beta \in L$.

The distinction between safety and liveness was motivated by different proof techniques used to validate them [21]. In general, safety properties can be proved by an inductive argument involving an invariant over the state of the system. Liveness properties are proved using well-foundedness arguments involving a termination function. Alpern and Schneider [2] have shown that every property (defined as a set of traces) can be written as the intersection between a safety property and a liveness property.

2.3 Asynchronous Systems

Systems with fair interleaving have a close relationship to *asynchronous systems*. The main advantage of asynchronous systems is that they can be characterized more by *non-assumptions* than by assumptions [32]: In asynchronous systems there is no assumed or existing bound on the relative processing speeds of processes. This means that while one process takes a single step, any other process can take an arbitrary (but finite) number of steps. In asynchronous systems where communication is through sending and receiving messages, usually also channels are assumed to be asynchronous [17] meaning that there is no upper bound on the time it takes for the system to deliver a sent message. Because they are so simple, the asynchronous system model has been used as the basis for many investigations in distributed algorithms.

2.4 Modeling Faults and Attacks

In the context of security we need to model faults and attacks. This is the basis for validating a certain system formally. In this paper, we take the view that attacks and faults can be modeled in the same way and hence we will use the terms “fault” and “attack” synonymously. According to Rushby [31], this can be done either in a *calculational* or *specificational* way. In the calculational approach, faults are modeled as unwanted program transitions which are explicitly incorporated into the faulty program. In effect, these approaches “calculate” the effects of faults and see whether the resulting traces still satisfy the specification. For example, work by Arora, Gouda and Kulkarni [4, 5] falls into this catgory as does all the work on software-implemented fault-injection [20]. In the specificational approach, faults are modeled by “weakening” the interface specifications of subcomponents. This is commonly done in the classical literature on fault-tolerant distributed algorithms [7, 23]. In both fault-modeling approaches, the faults have the effect of potentially *adding behavior* to a system, i.e., more system executions are possible if no countermeasures are taken. Hence, a *system in the presence of faults* is the original system which is modified to allow faulty behavior.

In this paper, a *specification* is a property, i.e., a set of traces. A system satisfies a specification *in the presence of faults* if all traces of the system in the presence of faults satisfy the specification.

2.5 Fault Classifications

Given a specification consisting of a safety and a liveness property, faults can be classified according to the type of property which they directly endanger. For example, memory perturbations that can be the effect of cosmic rays in spacecraft may lead to a direct violation of the safety property. In contrast to that, (silent) crash faults of processes do not necessarily endanger the safety property of the system but rather the liveness property (e.g., a process which is required to terminate but crashes before terminating).

Fault assumptions (like memory perturbation and crash) usually come with a restriction on the number of times faults of this class can happen. For example, in the context of self-stabilization [16] memory perturbations are assumed to occur only finitely many times. Similarly, for crash faults there is usually an assumed upper bound on the number of processes which are allowed to crash. The two aspects of a fault assumption are usually called *local* and *global*. While the local fault assumption enables additional component behavior, the global fault assumption restricts component behavior again.

In a sense, the Byzantine fault assumption [23] can be regarded as the strongest possible combination of safety and liveness violating faulty behavior. Byzantine behavior is *arbitrary* behavior of at most t components in the system. Some weaker variants of the pure Byzantine fault assumption (like *non-cooperative* Byzantine [28]) have also been defined. Sometimes their assumptions rely on the use of cryptographic primitives, like the *authenticated* Byzantine model [23] which is used to increase the resilience of Byzantine agreement protocols. In this context, the arbitrary behavior of the Byzantine adversary is restricted to not being able to "guess" a cryptographic key which it has no access to. In the context of secure message-passing systems, the Byzantine fault assumption has been adapted to additionally encompass the message transport system. This has become known as the *Dolev-Yao* attacker assumption [15]. In this model, the corrupted components (processes) together with the message system are seen as the adversary, i.e., messages (even between two uncorrupted parties) can be arbitrarily delayed or lost. However, signatures of uncorrupted processes cannot be forged.

Note that usually it is trivial to maintain a safety specification in the presence of faults if only "liveness affecting" faults (like crashes) may occur. Hence, satisfying both safety and liveness in the presence of faults is important. In the remainder of this paper, we will assume the Dolev-Yao model and investigate the role which liveness plays in the analysis of security protocols in this model.

3 Adapting Liveness in the Context of Denial of Service

3.1 Motivation

"Denial of Service" (DoS) attacks are a well-known threat to the availability of systems and these types of attacks have been widely experienced on the Internet. For example, a DoS attack in early 2000 seriously disrupted the services of some prominent Internet sites such as Amazon, eBay and Yahoo [13]. According to the CERT Coordination Center [12], a DoS attack "is characterized by an explicit attempt by attackers to prevent legitimate users of a service from using that service." This can be performed in a multitude of ways, including flooding the network, thereby preventing legitimate network traffic.

3.2 The Difficulty of Defining Denial of Service

The definition of CERT [12] can be regarded as seeing the "absence of liveness" as a definitory result of a DoS attack. Apart from compromising the availability of

the service, a successful DoS attack may wreak other forms of havoc, like a server operating system crash. These unwanted conditions can be incorporated into the safety specification of the system, and so a first approximation to formally defining DoS tolerance seems to be the following:

Definition 1 (tentative definition of DoS tolerance). *Given a system with a safety specification S and a liveness specification L in the Dolev-Yao model. The system is* DoS tolerant *if it satisfies S in the presence of faults.*

Since L is not required to be satisfied in the presence of faults, the system may lose all forms of liveness if it is attacked. This is similar to making satisfaction of liveness dependent on the behavior of the adversary. For example, Cachin et al. [8] define the liveness requirement of a validated Byzantine agreement protocol as follows:

> If all honest servers have been activated on [a certain instance of agreement] and all associated messages have been delivered [by the adversary], then all honest servers have decided [...].

Unfortunately, a definition in the spirit of Definition 1 is too weak to be useful since there are trivial implementations that tolerate DoS, namely systems that do nothing. However, a useful definition of DoS tolerance should at least contain Definition 1, since safety should be maintained at all times. For example, this allows to prohibit system crashes or other unpleasant consequences from excessive system load which usually are the effect of distributed DoS attacks.

The weakness of Definition 1 stems from its inability to reflect the behavior of practical DoS tolerant systems. In such systems, countermeasures are taken to prevent damage caused by high system loads. For example, the system can instruct a firewall or router to dismiss certain network traffic from malicious machines. If even this is not possible, the machine can cut off all its network connections alltogether (by shutting down the network interface). As a last resort, the system may cease operation alltogether by shutting down in a safe state. From these descriptions it should be obvious that real systems that tolerate DoS do not lose all forms of liveness in the presence of faults. They are still able to make a certain (limited) amount of progress, a form of "self-controlled" liveness.

3.3 Self-controlled Liveness

For a given labelled transition system with a set of actions A, define A_p as the set of actions *controlled by process p*. Formally, A_p includes all actions of p which have preconditions defined only over the local state of p. This means, process p can execute the actions in A_p independently of other processes in the system.

Definition 2 (self-controlled liveness). *A* self-controlled liveness property for process *p is a property L such that for all finite traces α exists a trace β such that $\alpha \cdot \beta \in L$ and β consists only of actions from A_p.*

Using Definition 2 we can now define DoS tolerance as follows:

Definition 3 (DoS tolerance). *Given a system with a safety specification S and a liveness specification L in the Dolev-Yao model and let L_p be the largest self-controlled liveness property contained in L. The system is* DoS tolerant for process p *if p satisfies S and L_p in the presence of faults.*

3.4 Relation to Machine Closure

We note here that there exists an interesting relation between Definition 2 and a concept from concurrency theory. Definition 2 is very close to the notion of *machine-closure* [25] (sometimes also called *feasibility* [3] or *machine-realizability* [1]). A liveness property L is machine-closed for a particular system iff for any finite trace its continuation demanded by L is a trace of the system. Metaphorically, this was characterized as the inability for a system "to paint itself into a corner" [34]. Self-controlled liveness properties are a specialization of machine-closure restricted to a subset of program actions. In the context of secure systems and DoS, this concept can therefore be helpful to characterize the ability to operate under the "progress restrictions" of an adversary. The relations between the different forms of liveness properties are depicted in Figure 1.

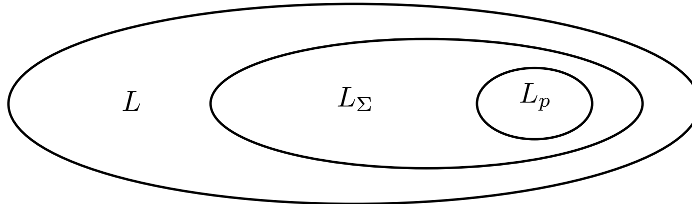

Fig. 1. Relations between all liveness properties, L, liveness properties which are machine-closed with respect to a system Σ, L_Σ, and self-controlled liveness properties for a particular process p from Σ, L_p.

3.5 Relation to Fail-Awareness

Recently, Cristian and Fetzer [14] introduced the *timed asynchronous system model*. This model is at heart an asynchronous system model. It contains, however, an explicit notion of real-time through the assumption that processes have access to local hardware clocks and that these clocks run within a linear envelope of real-time. This means that it is always possible to state a real-time bound on the maximum clock difference between each pair of clocks. Note that this does not mean that bounds exist on processing speeds or message delivery delays. On the contrary, the model explicitly assumes that communication is via message passing and there is no bound on message delivery delay. Just like in the time-free model, there is no bound on the relative processing speed of processes.

Through the notion of real-time provided by the hardware clocks it is possible to define real-time bounds for all services provided in a timed asynchronous system. In fact, this part of a service specification is mandatory. This means that while it is still impossible to, for example, reliably detect a process crash, it is now possible to tell whether a reply has not met its real-time deadline. This is called *fail-awareness* [18]. Since there are no bounds on fault occurrences, in this model it is not possible to ensure any liveness property at all.

To address the question of liveness, Cristian and Fetzer made the following observation: In practice, systems alternate between long periods of stability and short periods of instability. The measurements they give in their article [14] which were made in a local area network environment show that the average distance between unstable periods is 218 seconds, while the average length of an unstable period was about 340 milliseconds (this gives a ratio of 641 : 1). This observation allows to formulate *progress assumptions* of the form: "There exists a constant c, such that infinitely often there will be a stable period of length at least c." In other words, this means that infinitely often the system will be synchronous for at least c time.

Using progress assumptions it is now possible to specify liveness properties in the following way: "Assuming that some stability predicate holds, then the system will eventually perform an action." The stability predicate S can be, e.g., "infinitely often a majority of processes is synchronous for at least 2 seconds." Liveness properties of form L are transformed into weaker variants of the form $S \Rightarrow L$ called *conditional timeliness properties* [14].

In contrast to self-controlled liveness properties, conditional timeliness properties are restrictions on the *global* asynchrony, i.e., restrictions on the ability of the adversary to choose the scheduling of processes. Low-level timing assumptions can therefore be made explicit which are usually not expressible in the basic asynchronous system model. Self-controlled liveness properties restrict *local* "asynchrony", i.e., the ability of the adversary to stop an alive process. In this sense, self-controlled liveness properties can be regarded as a specialization of conditional timeliness properties.

4 Adapting Liveness in the Context of (Public-Key) Cryptography

4.1 Motivation

In the Dolev-Yao attacker model [15] all corrupted parties and the complete message system are assumed to be under the control of the adversary. However, it is often assumed that channels are secure and authentic meaning that messages between uncorrupted parties which are delivered by the adversary can be verified as being authentic and their message content remains secret. As discussed in Section 3, reactive protocols which are driven by messages cannot satisfy (general) liveness properties in this context since the adversary can alter, inject and schedule messages at its own choice. Interestingly, in this model it is possible to achieve liveness while losing security, as we now explain.

Systems involving cryptography are naturally prone to attacks based on cryptanalysis or on brute-force calculations to retrieve a secret piece of information. Now consider a system which satisfies a particular liveness property, say, termination. Since liveness properties do not state anything about the time it takes to achieve them, the adversary can delay the termination event as long as necessary to break the cryptographic keys involved in the protocol. This is one deficiency of the liveness property abstraction which only becomes apparent in the context of secure systems. In this section we review an interesting system model of Cachin, Kursawe, and Shoup [11, 22] (with extensions [8, 9]) from this perspective. The model introduces complexity theoretic means to rigorously analyse the security of a randomized Byzantine agreement protocol which is implemented using cryptography. Since in this model the cryptographic view of security prevails, the aim of this section is to explain it in a terminology related more to distributed systems and formal verification people.

4.2 Turing Machines, Security Parameters and Negligible Functions

Instead of transition systems, the model uses *probabilistic interactive Turing machines* (PITMs) to model individual processes. The reason for this is that, since they can read input from a dedicated input tape, PITMs have well-defined complexity measures with respect to their input. Theoretically there is no difference between a PITM and a parametrized transition system.

The input of such a PITM consists of a *security parameter*. This is also a concept rather unkown to regular distributed systems people. Briefly spoken, the security parameter k can be thought of as the "strength" of the underlying cryptography (e.g., the length of the secret key in bits). Attacks on the cryptosystem are assumed to have a running time which takes more than polynomial time in k. For example, a brute force attack on a cryptosystem with k key bits takes 2^k time. The hope is that by increasing k it is easily possible to combat the increasing processing power of new equipment. By having the PITMs read k from the input tape, the system model is parametrized in k.

Even though the complexity of a brute force attack increases exponentially in k, there is still a non-zero chance of an attacker simply "guessing" the right key. In practice, this chance is assumed to be negligible for the most common cryptosystems. For example, the chances of guessing an RSA key decreases exponentially in the length of the key, since all known algorithms for factoring large numbers need exponential time. Hence, the probability of guessing the right key decreases faster than any polynomial. This is formalized using the concept of a negligible function in k. A function $f(k)$ is called *negligible* if for all $c > 0$ there exists a k_0 such that $f(k) < \frac{1}{k^c}$ for all $k > k_0$. Hence, a negligible function decreases faster than any polynomial.

The adversary (which includes all corrupted processes as well as the message subsystem) is also modelled as a PITM with a time complexity bounded by a polynomial in k. The honest (i.e., uncorrupted) processes are considered to be "message driven" by the adversary, i.e., they only take one initial step (and generate a finite set of messages) and then only take steps whenever a message is

delivered to them by the adversary. In this case they perform a state transition and generate a (possibly empty) set of messages which are inserted into the message subsystem again.

4.3 The Liveness Property

Similar to restricting the liveness property in the presence of crash faults to all alive processes, any liveness property of a message-driven protocol in the context of the Dolev-Yao attacker model must be dependent on the extent to which the adversary delivers messages. For example, the liveness property of consensus from the introduction of this paper would be reformulated as:

- (Liveness) If all messages associated to a particular instance of consensus have been delivered, then all uncorrupted processes eventually decide.

Any other form of liveness would be trivially impossible to implement without dedicated resources. However, as motivated above, these properties alone do not capture the intuition of not being able to delay the protocol arbitrarily.

4.4 The Efficiency Property

Because the model is used to analyse the security of a protocol using complexity theory, runs necessarily need to be finite. The basic assumption of the model therefore is that any algorithm runs in a time which is polynomial in k. Hence, the length of any trace generated by an individual process in the system is bounded by some polynomial in k. For distributed systems people this may seem unusual at first since the notion of reactive systems was introduced explicitly to model non-terminating tasks like operating systems or schedulers. However, even "non-terminating" tasks terminate in practice, and because k and the polynomial can be freely chosen, the running time can be large enough to meet these facts.

Since processes are probabilistic, successive invocations of the system will usually generate different traces. For a given trace it is possible to define some complexity measure. For example, the *communication complexity* of a trace is the total bit length of all messages generated by honest processes during the trace. The communication complexity of the protocol consequently is a random variable that depends on the adversary and on k.

Given a particular protocol, it is possible to define a *protocol statistic* X which is a collection of random variables $\{X_A(k)\}$ for different adversaries A and different security parameters k. One member of this collection is obtained by measuring a particular complexity measure (like communication complexity) running the protocol with a particular adversary A and a particular security parameter k. The protocol statistic can therefore be seen as an abstraction of the behavior of the protocol for all adversaries and all security parameters (remember that only adversaries are allowed that run in polynomial time in k).

The idea now is to give a definition of what it means for a protocol to be "bounded" for all allowed adversaries and security parameters. This definition

can then be instantiated for different complexity measures. Intuitively, the complexity of a "bounded" protocol should always be bounded by a polynomial, no matter how the adversary operates.

Formally, a protocol statistic X is *uniformly bounded* if there exists a fixed polynomial $p(k)$ such that for all adversaries A there is a negligible function $e(k)$ such that for all $k > 0$ holds:

$$\Pr[X_A(k) > p(k)] \leq e_A(k) \tag{1}$$

This means that the probability that the complexity of the protocol lies above a certain fixed polynomial is negligible. Note that this holds for all (allowed) adversaries and security parameters.

Using the notion of uniformly bounded protocol statistic it is now possible to define an additional *efficiency* property for a protocol.

Definition 4 (efficiency property). *The communication complexity of the protocol is uniformly bounded.*

If the protocol in question is not randomized, it is possible to simplify Formula 1 and also Definition 4 to state that the communication complexity (dependent on k) should be below a fixed polynomial $p(k)$. To explain the intuition behind the definition, it should be instructive to look at a protocol which does not satisfy Definition 4 in the simplified (non-probabilistic) setting. Take for example a protocol with two types of messages: a-messages and b-messages. In the protocol a process, upon receiving an a-message, sends $n+1$ messages in reply to other processes which need to be processed in order to terminate (n being the number of processes in the system): n a-message and a b-message. The adversary can now hold back the b-message and deliver the a-message to generate additional messages. By repeating this procedure, the adversary can generate an exponential number of "correct" protocol messages in a time linear in k. Hence, the communication complexity for this particular adversary can surmount any polynomial in k. So this protocol does not satisfy the efficiency condition.

Intuitively spoken, the efficiency property ensures that the protocol terminates "fast" with respect to the extent to which the adversary delivers messages. The only assumption on the adversary is that its computing power is polynomially bounded. Hence, if a protocol satisfies efficiency then such a polynomially bounded adversary cannot delay the protocol in a superpolynomial way. But a superpolynomial delay is needed for a successful brute-force attack on the given cryptosystem. We can compare this setting metaphorically to a race with two competitors (protocol and adversary) and take the running time to be the measure of complexity. Both start running and the protocol wins if itself finishes in polynomial time. The adversary wins if the protocol takes superpolynomial time. The efficiency condition therefore ensures that the protocol always wins.

5 Conclusions

In the area of secure systems we are experiencing the development of an increasingly flexible formal machinery to help designing and validating them. We have

presented two concepts which we feel help to formalize and understand system properties in the presence of two different attacks: denial-of-service and brute-force attacks on cryptographic keys. We have argued that the security properties involved can be formalized in a way which builds on the well-established concepts of safety and liveness.

We feel that it has many methodological advantages to keep the framework of safety and liveness at the heart of any security investigations and find extensions in areas that fall outside of this domain. For example, distinguishing between (adversary-dependent) liveness properties and the efficiency property (as done in Section 4) allows the following: Security protocols can first be specified and analysed in the usual context of safety and liveness. If this is done, the efficiency of the protocol can then be investigated seperately. Hence, studying the resilience of protocols to brute-force attacks is compatible to the established methodology of verifying safety and liveness. This point is worth noting because other work has also developed methods to "incrementally" reason about properties that fall out of the safety/liveness domain. To the best of our knowledge, this has been done for information-flow properties in the context of non-interference [27] and real-time properties in the context of fault-tolerant algorithms [19, 26]. The goal is to continuously extend the collection of these analysis methods to tame the complexity of system validation by compositional reasoning.

Acknowledgments

The author wishes to thank Klaus Kursawe for his insightful explanations on the motivations of his work. Thanks also to Heiko Mantel, Michael Waidner, Holger Vogt and Hagen Völzer for helpful discussions.

References

1. Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.
2. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
3. Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.
4. Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
5. Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.
6. Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.
7. Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.
8. Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS-9)*, Washington, DC, 2002.

9. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology – CRYPTO ' 2001*, Lecture Notes in Computer Science. International Association for Cryptologic Research, Springer-Verlag, 2001. See [10] for long version.
10. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. Record 2001/006, Cryptology ePrint Archive, January 2001. An extended abstract was published as [9].
11. Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 123–132, Portland, Oregon, 2000.
12. CERT Coordination Center. Denial of service attacks. Internet: `http://www.cert.org/tech_tips/denial_of_service.html`, June 2001.
13. CNN.com. Cyber-attacks batter web heavyweights. Internet: `http://www.cnn.com/2000/TECH/computing/02/09/cyber.attacks.01/index.htm%l`, February 2000.
14. Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.
15. Danny Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.
16. Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
17. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
18. Christof Fetzer and Flaviu Cristian. Fail-awareness: An approach to construct fail-safe applications. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 282–291. IEEE, June 1997.
19. Jean-François Hermant and Gérard Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, August 2002.
20. Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.
21. Ekkart Kindler. Safety and liveness properties: A survey. *EATCS-Bulletin*, (53), June 1994.
22. Klaus Kursawe. Asynchronous byzantine group communication. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS), Workshop on Reliable Peer-to-Peer Distributed Systems*, pages 352–357, Osaka, Japan, October 2002. IEEE Computer Society Press.
23. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
24. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.
25. Leslie Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.
26. Gerard Le Lann. On real-time and non real-time distributed computing. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, pages 51–70, September 1995.
27. Heiko Mantel. Possibilistic definitions of security - An assembly kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW 2000)*, Cambridge, England, July 2000. IEEE Computer Society Press.

28. Asif Masum. *Non-cooperative Byzantine failures: A new framework for the design of efficient fault tolerance protocols*. PhD thesis, Universität-Gesamthochschule Essen, Fachbereich Mathematik und Informatik, 2000. Published by Libri Books on demand, ISBN 3-8311-0815-3.
29. Catherine Meadows. Open issues in formal methods for cryptographic protocol analysis. In *DISCEX 2000*, pages 237–250. IEEE Computer Society Press, January 2000.
30. Inc. RSA Data Security. Rsa code-breaking contest again won by distributed.net and electronic frontier foundation (eff). Internet: `http://www.rsasecurity.com/company/news/releases/pr.asp?doc_id=462`, January 1999.
31. John Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.
32. Fred B. Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems*, chapter 2, pages 17–26. Addison-Wesley, Reading, MA, second edition, 1993.
33. John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8–17, June 1992.
34. Hagen Völzer. *Fairness, Randomisierung und Konspiration in verteilten Algorithmen*. PhD thesis, Humboldt Universität zu Berlin, Fakultät für Informatik, December 2000.