Innovations Syst Softw Eng (2006) 2: 39–48
DOI 10.1007/s11334-006-0023-7

ORIGINAL PAPER

Ali Mili · Frederick Sheldon · Lamia Labed Jilani
Alex Vinokurov · Alex Thomasian · Rahma Ben Ayed

Modeling security as a dependability attribute: a refinement-based approach

Received: 10 July 2005 / Accepted: 13 October 2005 / Published online: 24 February 2006

Abstract As distributed, networked computing systems become the dominant computing platform in a growing range of applications, they increase opportunities for security violations by opening hitherto unknown vulnerabilities. Also, as systems take on more critical functions, they increase the stakes of security by acting as custodians of assets that have great economic or social value. Finally, as perpetrators grow increasingly sophisticated, they increase the threats on system security. Combined, these premises place system security at the forefront of engineering concerns. In this paper, we introduce and discuss a refinement-based model for one dimension of system security, namely survivability.

1 Introduction: motivation

The term *missile gap* was coined in the late 1950s when the Soviets launched Sputnik, to refer to the wide gap between the US's national capabilities in space and its national aspirations. The term *software gap* was coined in the mid to late 1980s, at the height of the IT revolution, when it was clear that software technology was not keeping up with the demands of the world economy as it was growing increasingly dependent on the safe, reliable operation of software systems. We coin the term *security gap* to refer to the vast technological gap that exists today between available capabilities and the demands imposed by recent global developments. We submit that the security gap matches or exceeds the earlier gaps in terms of what is at stake, and in terms of its technical challenge.

In this section we discuss in turn: the need for modeling security, then the need for modeling security as an attribute of dependability, and finally the adequacy of a refinement-based approach to modeling security. These will be the subject of the next three subsections.

1.1 Modeling security

Even though logically, system reliability is driven exclusively by the existence and possible manifestation of faults, empirical observations regularly show a very weak correlation between faults and reliability. In [22], Mills and Dyer discuss an example where they find a variance of 1–50 in the impact of faults on reliability; i.e. some faults cause system failure 50 times more often than others; while their experiment highlights a variance of 1–50, we argue that actual variance is in fact unbounded. Also, they find that they can remove 60% of a system's faults and improve its reliability by only 3%. [1] In a study of IBM software products, Adams [1] finds that many faults in the system are only likely to cause failure after hundreds of thousands of months of product usage.

We argue that the same may be true for security: vulnerabilities in a system may have widely varying impacts on system security. In fairness, the variance may be wider

[1] Given that typically system-level testing consumes nearly 50% of lifecycle costs and hardly comes close to discovering 60% of system faults, this finding is a resounding condemnation of random fault-chasing, and advocates instead a discipline that leads us towards the most influential faults first.

A. Mili (✉) · A. Vinokurov · A. Thomasian
College of Computing Science,
New Jersey Institute of Technology,
Newark, NJ 07102-1982, USA
E-mail: mili@cis.njit.edn

A. Mili
Software Engineering Institute,
Carnegie Mellon University,
Pittsburgh, PA 15213, USA

F. Sheldon
U.S. DOE Oak Ridge National Lab,
PO Box 2008, MS 6085,
1 Bethel valley Road,
Oak Ridge, TN 37831-6085, USA

L. L. Jilani
Institute of Management,
University of Tunis,
Bardo 2000, Tunisia

R. B. Ayed
School of Engineering,
University of Tunis,
Belvedere 1002, Tunisia

for reliability than for security, because in malicious security violations high-impact vulnerabilities may be more attractive targets than lower-impact vulnerabilities, but wide variances are still quite plausible. Wide variances, to the extent that they are borne out, have broad impacts on security management:

- In practice, security ought not be defined as the absence of vulnerabilities, no more than reliability is defined by the absence of faults (low-impact vulnerabilities do not affect security in a meaningful way).
- In practice, security ought not be measured or quantified by the number of vulnerabilities, just as it is widely agreed (as highlighted by the works of Adams [1] and Mills [22]) that faults per KLOC (Kilo Lines of Code) is an inappropriate measure of reliability. Though fault density is commonly used as a measure of programmer productivity and product quality, it has long been discredited as a measure of reliability.
- Security cannot be improved by focusing on vulnerabilities, as we have no way to tell whether a given vulnerability has low (1) or high (50) impact on security. Rather, security should be managed by pursuing a policy that leads us to the highest-impact vulnerabilities first (a similar approach to usage pattern testing [3,10,15–17,22,23,25]).

In light of these observations, we argue in favor of modeling security in a way that reflects its visible, measurable, observable attributes, rather than its hypothesized causes. To this effect, we introduce the tentative outline of a *logic for system security*, which represents/captures security properties in terms of its observable attributes. This logic is defined in terms of the following features:

- A notation for *security specification*, which details how to capture security requirements of a system.
- A formula for *security certification*, which formulates the condition under which a system (represented by its security abstraction) meets a given set of security requirements (represented by security specifications).

Note that, in order to quantify reliability as the mean time to *failure*, we must define what it means to *fail*, which in turn requires that we define *specification* and *correctness*. Likewise, defining and quantifying security requires that we define the concepts of security specification and security certification. In this paper, we discuss broad premises that characterize our approach, and present tentative notations and formulas for the proposed logic for system security.

1.2 Security as a dimension of dependability

It is customary [27] to define *dependability* as the aggregate of four attributes: *availability* (probability of providing services when needed), *reliability* (probability of failure-free operation), *safety* (probability of disaster-free operation) and *security* (probability of interference-free operation). Like most classifications, this one does not define sharp distinctions between these attributes, but is convenient because it highlights meaningful differences. An important distinction is between *availability*, which deals with operational properties, and the other three, which deal with functional and behavioral properties. We argue in favor of a uniform model to capture the three behavioral properties, i.e. reliability, safety and security. We submit three broad arguments to support our position.

- *Conceptual argument*. Generalization is a problem-solving strategy that substitutes a specific problem with a more general problem, thereby abstracting away many irrelevant problem-specific details; though it is paradoxical, it is an effective problem-solving strategy, as it tends to abstract away irrelevant detail, and thereby produce more elegant solutions. We argue that reliability, safety and security lend themselves to analysis by generalization, by virtue of their commonalities. Reliability is defined in terms of faults, errors and failures, and is handled by means of a hierarchy of three methods: fault avoidance, fault removal and fault tolerance. Safety is defined in terms of three concepts, hazard, mishap and accident, and is handled by means of a hierarchy of three methods: hazard avoidance, hazard removal and damage limitation. Security is defined in terms of three concepts, vulnerability, threat and exposure (or attack), and is handled by means of a hierarchy of three methods: vulnerability avoidance, vulnerability neutralization, and exposure limitation. We argue that these analogies are a strong hint to at least attempt to model these attributes in a uniform manner.
- *Pragmatic argument*. Reliability, safety and security are interdependent, in the sense that whether each property holds may depend on the others. For example, all the claims of reliability and safety become void if an intrusion occurs and alters the system's function or state. Conversely, the security of a system is dependent on the reliability of the components that implement/enforce its security measures. Hence in practice, having high values for one of these attributes is probably not meaningful unless we have commensurate values for the other attributes as well; also, it is conceivable that the proof of any one of these properties will use hypotheses about the other properties. In addition, while the distinctions between reliability, safety and security are meaningful for the engineer, they are less meaningful for the user. From the standpoint of the user, it matters little whether a system failed because of violation of a reliability requirement, a safety requirement, or a security requirement. Hence while the distinction between these properties may be convenient for the sake of discussion and characterization, in practice it is best to model and analyze these properties as a whole.
- *Methodological argument*. By virtue of the analogies that we highlighted above, it is very likely that methods developed for one property will prove to be useful to other properties. In particular, it is quite plausible that methods developed for ensuring reliability prove to be useful in ensuring security. We will explore this possibility in the sequel.

1.3 A refinement-based approach

Whereas in the previous subsection, we argued in favor of a uniform approach to the three behavioral attributes of dependability (reliability, safety, and security) in this subsection we argue in favor of a specific uniform approach, one based on refinement calculi. We submit three broad arguments to support our position.

- *Conceptual argument*. Refinement calculi have long been used to model correctness properties, and hence form the basis for reliability analysis [4,9,12–14,29]. Mili et al. [21] have shown how safety can be modeled by the same refinement mathematics, and how reliability and safety concerns can be addressed with the same recovery mechanism. In this paper we submit that the same refinement mathematics can be used to model (some aspects of) security, and will discuss ways to do so in the sequel. In particular, we argue that there is no difference between reliability and safety except in the quantification of failure costs (the violation of a safety requirement costs more than the violation of a reliability requirement). We will also argue that there is no difference between reliability and security except in the characterization of fault hypotheses (reliability deals with hardware or software faults while security deals with faults caused by malicious intervention). We will elaborate on these ideas in the sequel.
- *Pragmatic argument*. Nicol et al. [24] submit that it is virtually impossible to ascertain the security of a complex system, and argue that in order to enhance system security we must deploy a wide range of methods:

  > "What is needed is an integrated validation framework that permits the use of multiple evaluation techniques in an organized manner. ... A symbiotic relationship should be established among the various techniques such that they complement and supplement each other to build the overall argument".

  We argue that some of the work we have done on reliability does exactly that, deploying a range of methods to maximize coverage and minimize (or at least control) cost [4,20]. More recently, we have explored means to extend this work to deal with security, by trying to cast security properties in a refinement-like model [5]. The purpose of this approach is to combine dependability measures irrespective of how they are analyzed and implemented. Specifically, we have developed mathematics that allow us to
  - *Decompose dependability goals* into simpler subgoals, that can be dispatched to distinct verification and validation efforts that may or may not use the same methods.
  - *Compose dependability claims* obtained from different methods deployed on different parts of a system dealing with different aspects of dependability.

  The ability to decompose goals and compose claims is an essential component of our strategy for dealing with large and complex systems.
- *Methodological argument*. By merging various aspects of dependability and modeling them in a uniform manner we allow ourselves to build eclectic arguments of dependability. We envision to store all dependability claims in a unique database, which we can query for specific properties of interest; details of this approach are discussed in the sequel.

1.4 Literature review

Nicol et al. [24] present an extensive survey of dependability models, and analyze from the perspective of applying them and extending them to security. In particular, they distinguish between three broad classes of models: combinatorial methods; model checking; and state-based stochastic methods. The extensive survey of Nicol et al. highlights the lack of work on extending existing refinement calculi to security modeling, which is our intent in this paper.

2 Genesis of an integrated approach

In [4,20] we consider the traditional classification of program or system-verification methods into three broad classes: *fault avoidance*, *fault removal* and *fault tolerance*. Also, we introduce a refinement-based logic that has the following features:

- Specifications and programs are represented by binary relations, and refinement is represented by an ordering between relations.
- The refinement relation is a partial ordering, which has some lattice-like properties. We represent the ordering by $\sqsupseteq$ and the lattice operators by $\sqcup$ (for *join*, i.e. least upper bound) and $\sqcap$ (for *meet*, i.e. greatest lower bound).
- While the meet is defined for any pair of specifications, the join is not. Only pairs of specifications that admit a common refinement admit a join. Also (perhaps consequently) the lattice of refinement has no universal upper bound, though it does have a universal lower bound.

We use this refinement logic to address two issues: how to compose verification claims that stem from distinct verification methods; and how to decompose a complex verification goal into simpler subgoals in such a way as to minimize or control overall verification costs. We briefly address these issues in the sequel, referring the interested reader to [4,20] for technical details.

2.1 Composing verification claims

Interestingly, we find that all three methods can be captured in this refinement logic by a formula of the form

$$S \sqsupseteq P$$

where S is the system under consideration and P is the specification (property) we are verifying about S. The form of P depends on the parameters of the method being used.

– For verification, P is the binary relation defined in terms of the precondition [say $\Phi(s)$] and post-condition [say, $\Psi(s_0, s)$] as follows:

$$P = \{(s, s') | \Phi(s) \wedge \Psi(s, s')\}.$$

We denote the right-hand side of this equation by V (for: verification).
– For testing, P is the relation defined in terms of the oracle (say Ω) and test data (say D) as follows:

$$P = \{(s, s') | s \in D \wedge (s, s') \in \Omega\}.$$

We refer to the right-hand side of this equation as T (for testing).
– For fault tolerance, P is defined in terms of the relations that represent error detection (E) and error recovery (R), as follows:

$$P = E \sqcap R.$$

We denote the right-hand side of this equation by F (for *fault tolerance*), and we justify it simplistically by the following argument: we assume that we are ensuring fault tolerance by means of an error-detection routine that checks for some binary condition E between a past and a current state, and eventually (if condition E is not satisfied) invokes recovery routine R that maps a past state onto a correct new state, according to the following (schematic/ simplified pattern):

```
if not E then R;
```

Because we do not know ahead of time whether E holds or not, we cannot tell whether relation E or relation R holds between the past state and the current state. Hence all we can claim is that the system under review refines E or R. By virtue of a lattice identity,

$$S \sqsupseteq E \vee S \sqsupseteq R \Rightarrow S \sqsupseteq (E \sqcap R)\,.$$

The lattice properties of the refinement ordering allow us to combine verification claims by virtue of the following identity:

$$S \sqsupseteq V \wedge S \sqsupseteq T \wedge S \sqsupseteq F \Rightarrow S \sqsupseteq (V \sqcup T \sqcup F)\,,$$

assuming V, T, and F admit a join. Hence if we have established (using a static verification method) that S refines V, that S refines T (using a dynamic testing method), and that S refines F (using a fault tolerance method), we can claim that S satisfies the aggregate specification

$$V \sqcup T \sqcup F\,.$$

2.2 Decomposing verification goals

Not only does the refinement lattice allow us to combine eclectic verification claims, but it also allows us to decompose complex verification goals, by virtue of the following observations:

– Complex specifications can naturally be decomposed as joins of simpler specifications [6].
– Lattice identities provide that, if a system refines all the terms of a join, then it refines the join.
– Perhaps most interesting of all is the observation that the effectiveness, ease of application, and reliability of a verification method vary a great deal according to the specification at hand. The same verification result, say

$$S \sqsupseteq P,$$

can be proved very easily, effectively, and reliably with one method (static verification, dynamic testing, or fault tolerance) yet at the cost of great difficulty and complexity with another method, depending on the properties of P. Specifically, static verification methods are most effective for reflexive transitive relations, because such methods are typically inductive, and reflexivity makes the basis of induction trivial while transitivity makes the induction step trivial. Dynamic testing methods are most effective for relations that can be implemented reliably as oracles (a faulty oracle may undermine the whole testing effort by giving misleading diagnoses). Fault-tolerance methods are most effective for unary relations (dealing exclusively with the current state) that can be implemented efficiently (to reduce computation overhead) and do not require saving previous state spaces (to reduce memory overhead as well).

In [20] we outlined a systematic procedure for analyzing relational specifications and assigning them appropriate verification methods so as to minimize overall verification effort and maximize trustworthiness. Also, we have illustrated this approach on a simple example, involving a Gaussian elimination program.

3 A uniform representation of dependability claims

Combining diverse methods to reach a uniform verification goal is a commendable approach, but it suffers from the following shortcomings:

– Verification methods are best captured by probabilistic statements rather than logical statement; even the most formal verification methods have a degree of uncertainty, which we do not model in our logical interpretation.
– All verification methods are based on (implicit) assumptions, which the logical model discussed above does not capture. For example, static verification is based on the assumption that the verification rules used in the proof are borne out by the compiler and run-time system on which the program runs. Also, dynamic testing is based on the assumption that the testing environment is consistent with (or is harsher than) the operating environment. Finally, fault-tolerance methods are usually based on the assumption that the fault tolerance (error detection, error recovery) routines are free of faults.

– Another issue with the logical interpretation is that it is too narrow: for example, we have interpreted the process of testing a program S against an oracle Ω using test data D by the formula

  $$S \sqsupseteq T\,,$$

  where T is the restriction of Ω to D. While strictly speaking that is all we can claim, in fact most often we test S against oracle Ω using data D to establish that S refines Ω, assuming that test data D is (somehow) a faithful representation of the whole input domain of the program. This interpretation is reflected by the (stronger) claim

  $$S \sqsupseteq \Omega.$$

  In the proposed new approach, we do not have to choose between these two interpretations; we can keep them both, provided we learn how to combine them.
– Absent from the logical interpretation is the concept of failure cost. We argue in favor of a model that quantifies failure costs associated with various situations; in particular, we argue that the main difference between reliability and safety is a difference of failure costs.

3.1 A probabilistic model

In light of the foregoing premises, we propose that all verification results be characterized by the following attributes:

– *Property*. This attribute reflects the property that we are claiming for the product under consideration. Possible values for this attribute include: correctness (refinement), operational attributes (such as response time, for example), recoverability preservation [21], security property (protection against intrusion, for example), etc. Overloading the $\sqsupseteq$ symbol, we will denote this attribute by $\sqsupseteq$ even though we refer more generally to any kind of property.
– *Reference*. This complements the previous attribute by specifying with respect to what specification we are claiming the property. In the case of correctness or recoverability preservation, for example, the reference in question would be the specification of relevant functional requirements. In the case of response time, for example, this would be the specification of relevant response-time requirements. In the case of a security property, this would be the specification of relevant security requirements (whose form we will explore subsequently). We usually represent this attribute by the symbol R.
– *Assumption*. As we discussed above, each verification method has an implicit assumption, which we highlight through this attribute; we can also use this attribute to highlight additional conditions. For example, if we test program S against oracle Ω and we use a *representative* data set D, we are assuming that the program is correct with respect to Ω if and only if it is correct with respect to the restriction of Ω to D; such a claim is contingent upon D being a representative set for the domain of the program. We usually represent this attribute by A (for assumption).
– *Certainty*. This attribute quantifies the probability we estimate for the verification claim being made; we usually denote it by p.
– *Stake*. Also referred to as a the *failure cost*, this attribute quantifies the cost of failing to satisfy the claimed property with respect to the claimed reference. This attribute allows us to reflect the fact that different requirements carry different stakes and that some may be more critical than others.
– *Expense*. Also referred to as the *verification cost*, this attribute quantifies the cost of verifying a particular claim; as we have discussed in Sect. 2, this attribute depends a great deal on the specification (reference) against which the claim is established.

To reflect all these attributes, we represent verification claims using the generic format:

$$\Pi(S \sqsupseteq R|A) = p,$$

which we read as: the probability that system S refines (in the general sense) specification (or reference) R under the assumption A is p. Furthermore, to capture failure cost and verification cost, we introduce two additional functions:

– The *failure cost function*, which maps a property and reference onto a cost. Formally,

  $$\nu\colon Prop \times Ref \rightarrow Cost.$$

– The *verification cost function*, which maps a property, reference, method, and assumption onto a cost. Formally,

  $$\mu\colon Prop \times Ref \times Meth \times Assum \rightarrow Cost.$$

  The verification cost clearly depends on the property to be verified and the method used for verification; as we discussed in Sect. 2, it also depends on the reference (specification); finally, it clearly depends on the assumption of the proof (the stronger the assumption, the easier the proof).

To illustrate/justify the proposed model, we use it to briefly represent some sample verification claims.

– *Testing experiment I*. If we test a program S on some test data D against some oracle Ω and find it to run correctly on all the elements of D, then we can claim

  $$\Pi(S \sqsupseteq T|A \wedge B) = 1.0\,,$$

  where T is the restriction of Ω to D, A is the assumption that the testing environment is equivalent to (or harsher than) the operating environment and B is the assumption that the oracle (and other test set-up code) is correct.
– *Testing experiment II*. If we test a program S on some test data D against some oracle Ω and find it to run correctly on all the elements of D, then we can claim (for example)

  $$\Pi(S \sqsupseteq \Omega|A \wedge B \wedge C) = p\,,$$

  where A and B are the assumptions defined above and C is the assumption that the test data is a faithful representative of S's input domain (i.e. if S succeeds on D then we can infer with probability p that it succeeds on all its domain), if and only if it succeeds on all its domain.

3.2 Integrating failure costs

In this section, we briefly discuss the impact of introducing failure costs into our model; in particular, we argue that failure costs enable us to propose a generic measure of system dependability. When we talk about a system's mean time to failure (MTTF), we usually do so with respect to two implicit parameters: First, an implicit specification; and second an implicit failure cost. It is only with respect to these two implicit parameters that the MTTF makes sense.

This approach can be generalized in three directions:

- First, complex specifications typically represent of a wide range of requirements, which the traditional concept of MTTF lumps into one.
- Second, different requirements may have widely varying failure costs, hence failing one requirement may mean something totally different from failing another.
- Third, the same requirement may carry different stakes for different stakeholders of a system, hence failure cost depends not only on the requirement but also on the stakeholder.

To illustrate this situation, consider a flight control system of a commercial airliner, for example. We can imagine the following stakeholders in the operation of such a system: a passenger; the pilot; the Federal Aviation Authority (FAA); the airline executive; the insurance company that insures the aircraft; etc. On the other hand, we could consider a wide range of (nonorthogonal, overlapping) requirements: that the ride be smooth; that the flight be fuel-efficient; that the flight be timely; that the flight follow its route within a few hundred feet; that the flight be safe; etc. It is easy to see how each stakeholder has different stakes in each requirement; this can be represented in a two-dimensional table (stakeholders versus requirements).

In a context like this, the MTTF of the system is not sufficiently informative, if it talks about a global failure to satisfy the specification, but does not distinguish between the various components of the specification. We argue instead in favor of a measure that acknowledges variations in stakeholders and in stakes, and reflects the *mean failure cost*. This function can be quantified by a monotonic combination (over requirements R_i) of terms of the form:

$$(1 - \Pi(S \sqsupseteq R_i)) \times \nu_j(\sqsupseteq, R_i)\,,$$

where ν_j is the cost function for stakeholder j (hence $\nu_j(\sqsupseteq, R_i)$ is the cost for stakeholder j of failure to satisfy requirement i). We talk about monotonic combination of these terms rather than their sum, because the requirements are not necessarily orthogonal or disjoint; the exact structure of this formula is currently under investigation.

This discussion illustrates in what sense reliability and safety can be modeled alike: If R_i is a safety requirement, then typically $\nu_j(\sqsupseteq, R_i)$ is very high, and therefore the term

$$(1 - \Pi(S \sqsupseteq R_i))$$

must be very low, in order to maintain a low value for the product

$$(1 - \Pi(S \sqsupseteq R_i)) \times \nu_j(\sqsupseteq, R_i),$$

To keep this value low, we must ensure with the greatest possible certainty that S refines R_i. From this standpoint, there is no distinction between safety and reliability; all we see are requirements with varying failure costs.

4 Towards a logic for system security

In this section, we analyze system security and investigate in what sense and how it can be folded into the uniform model discussed above for representing dependability claims.

4.1 Assigning meaning to security measures

Both safety and reliability can be represented by the claim that the system/program under consideration refines a given specification. We write this abstractly as

$$S \sqsupseteq R\,.$$

The first question that we wish to raise as we attempt to model security is: is security modeled with a different property ($\sqsupseteq$) or a different reference (R)? The answer, as we will discuss, is both.

Nicol et al. [24] discuss a number of dimensions of security, including: *data confidentiality*, *data integrity*, *authentication*, *survivability*, and *nonrepudiation*. In the context of this paper, we focus our attention on *survivability*, and readily acknowledge a loss of generality; other dimensions of security are under investigation. Survivability is defined in [11] as the capability of a system to fulfill its mission in a timely manner, in the presence of attacks, failures, or accidents [24]. We discuss in turn how to represent security (survivability) requirements, and how to represent the claim that a system meets these security requirements.

4.1.1 Specifying security requirements

We note that there are two aspects to survivability: the ability to deliver some services, and the ability to deliver these services in a timely manner; to accommodate these, we formula security requirements by means of two relations, one for each aspect. Using a relational specification model presented in [6], we propose to formulate functional requirements as follows:

- An input space, that we denote by X; this set contains all possible inputs that may be submitted to the system, be they legitimate or illegitimate (part of an attack/intrusion).

– Using the space X, we define the space H, which represents the set of sequences of elements of X; we refer to H as the set of *input histories* of the specification. An element h of H represents an input history of the form

$$..h_N.h_{N-1}\ldots h_3.h_2.h_1.h_0,$$

where h_0 represents the current input, h_1 represents the previous input, h_2 represents the input before that, etc.
– An output space Y, which represents all possible outputs of the system in question.
– A relation ϕ from H to Y that specifies for each input history h (which may include intrusion/attack actions) which possible outputs may be considered correct (or at least acceptable). Note that ϕ is not necessarily deterministic, hence there may be more than one output for a given input history. Note also that this relation may be different from the relation R that specifies the normal functional requirements of the system: while R represents the desired functional properties that we expect from the system, ϕ represents the minimal functional properties we must have even if we are under attack; hence while it is possible to let $\phi = R$, it is also possible (perhaps even typical) to let there be a wide gap between them.

As for representing timeliness requirements, we propose the following model:

– The same input space X, and history space H.
– A relation from H to the set of positive real numbers, which represents for each input history h the maximum response time we tolerate for this input sequence, even in the presence of attacks. We denote this relation by ω.

In the sequel, we discuss under what conditions do we consider that a system S satisfies the security requirements specified by the pair (ϕ, ω).

4.1.2 Certifying security properties

Given a security requirements specification of the form (ϕ, ω), we want to discuss under what conditions we consider that a program S that takes inputs in X and produces outputs in Y can be considered to satisfy these security requirements. Space limitations preclude us from a detailed modeling of attacks/intrusions, hence we will, for the purposes of this paper, use the following notations:

– Given a legitimate input history h, we denote by $\upsilon(h)$ an input history obtained from h by inserting an arbitrary intrusion sequence (i.e.. a sequence of actions that represent an intrusion into the system).
– Given an input history h (that may include intrusion actions) we denote by $\theta(S, h)$ the response time of S to the input history h.

Using these notations, we introduce the following definition.

Defintion 1 *A system S is said to be secure with respect to the specification (ϕ, ω) if and only if*

1. *For every legitimate input history h,*
$$(h, S(h)) \in \phi \Rightarrow (\upsilon(h), S(\upsilon(h))) \in \phi.$$
2. *For every legitimate input history h,*
$$\theta(S, h) < \omega(h) \Rightarrow \theta(S, \upsilon(h)) < \omega(h).$$

The first clause of this definition can be interpreted as follows: if system S behaves correctly with respect to ϕ in the absence of an intrusion, then it behaves correctly with respect to ϕ in the presence of an intrusion. Note the conditional nature of this clause: we are not saying that S has to satisfy ϕ at all times, as that is a reliability condition; nor are we saying that S has to satisfy ϕ in the presence of an intrusion, as we do not know whether it satisfies in the absence of an intrusion (surely we do not expect the intrusion to improve the behavior of the system; all we hope for is that it does not degrade it). Rather, we are saying that, if S satisfies ϕ in the absence of an intrusion, then it satisfies it in the presence of an intrusion.

The second clause articulates a similar argument, pertaining to the response time: if the response time of S was within the boundaries set by ω in the absence of an intrusion, then it remains within those bounds in the presence of an intrusion.

The definition that we propose here is focused entirely on effects rather than causes, and gives meaning to the concept of *security failure*. Using this concept, we can now quantify security by the same *MTTF*, where *F* stands for security failure. Stevens et al. [28] present measures of security in terms of mean time to vulnerability discovery (MTTD) and mean time to exploitation of discovered vulnerability (MTTE). By contrast with our (re-)definition, these definitions are focused on causes (rather than effect); in fairness, Stevens et al. [28] propose them as intruder models rather than security models. The difference between our effect-based measure and Stevens's cause-based measure is that a vulnerability may be discovered without leading to an intrusion, and an intrusion may be launched without leading to a security failure in the sense of our definition.

4.2 Inference support

In the light of these discussions, we can now represent security claims in the same notation proposed earlier for other dimensions of dependability, i.e., as

$$\Pi(S \sqsupseteq R|A) = p\,,$$

but with a qualification: R represents security requirements, as discussed in Sect. 4.1.1 and $\sqsupseteq$ represents the security property, as we have defined it in Sect. 4.1.2. Cost functions can be added to quantify values of failure cost and verification cost.

We have developed a *very* sketchy prototype of a tool that stores claims and supports queries. We envision several types of inference rules that such a tool can deploy: probabilistic rules (that use identities of probability calculi); refinement-based rules (that use the ordering properties of the refinement relation); lattice-based rules (that use identities of lattice theory); rules that stem from the interrelationships between the

various dimensions of dependability (for example, that security depends on the reliability of the security components); etc.

In its current form, the prototype includes only *probability rules*, and hence has very limited capability. Nevertheless, it allows us to discuss our vision of its function and its operation. The first screen of the prototype offers the following options:

- *Record a reliability/safety claim*. Clicking on this tab prepares the system to receive details about a dependability claim (reliability, safety, etc.) with respect to a functional specification. Given that such claims have the general form:

  $$\Pi(P \sqsupseteq R|A) = p,$$

  the system prompts the user to fill in fields for the property ($\sqsupseteq$), the reference (R), the assumption (A), and the probability (p).
- *Record a security claim*. Clicking on this tab presents an entry screen that prompts the user for a security specification (two fields: a functional requirement and an operational requirement), a field for an assumption, and a field for a probability. There is no need for a *property* field, since the property is predetermined by the choice of tabs.
- *Record cost information*. As we recall, there are two kinds of cost information that we want to record: failure cost, and verification cost. Depending on the user's selection, the system presents a spreadsheet with four columns (property, reference, cost, and unit—for failure cost), or six columns (property, reference, method, assumption, cost, and unit—for verification cost). This information is stored in tabular form to answer queries subsequently on failure costs or verification costs.
- *Record domain knowledge*. Because dependability claims are formulated using domain-specific notations, a body of domain-specific knowledge is required to highlight relevant properties and relationships, and to enable the inference mechanism to process queries. This domain knowledge is recorded by selecting the appropriate tab on the system.
- *Queries*. Clicking on the tab titled *submit query* prompts the user to select from a list of query formats. The only format that is currently implemented is titled *validity of a claim*, and its purpose is to check the validity of a claim formulated as

  $$\Pi(P \sqsupseteq R|A) \geq p,$$

  for some property $\sqsupseteq$, reference (specification) R, assumption A, and probability p. Notice that we do not have equality, but inequality; this feature can be used if we have taken a number of dependability measures and wish to check whether they are sufficient to allow us to claim that P refines R with a greater certainty than a threshold probability p.

To answer a query, the system composes a theorem that has the query as goal clause, and uses recorded dependability claims and domain knowledge as hypotheses. The theorem prover we have selected for this purpose is *Otter* [18,19,30].

4.3 A sample demo

To illustrate the operation of the tool, we take a simple example. We will present, in turn, the dependability claims that we submit to this system, then the domain knowledge, and finally the query; this example is totally contrived and is intended only to illustrate what we mean by composing diverse dependability claims. Also, even though the model that we envisage has inference capabilities that are based on many types of rules (probabilistic identities, refinement rules, lattice identities, relations between various refinement properties, etc.), in this demo we only deploy probabilistic rules.

For the purposes of this example, we summarily introduce the following notations, pertaining to a fictitious nuclear power plant:

- *Specifications*. We consider a specification, which we call *SafeOp*, which represents the requirement that the operation of the reactor is safe. We also (naively) assume that this requirement can be decomposed into two subrequirements, whose specifications, *CoreTemp* and *ExtRad*, which represent requirements for safe core temperatures and safe external radiation levels.
- *Assumptions*. We assume (artificially) that the claims we make about refining the specifications *CoreTemp* and *ExtRad* are contingent upon a combination of conditions that involve two predicates: *FireWall*, which represents the property that the system's firewall is operating correctly; and *ITDetection*, which represents the property that the system's insider threat detection is working properly.

Using these notations, we illustrate the deployment of the tool by briefly presenting the security claims, the domain knowledge, then the query that we submit to it.

- *Claims*. Using the system's GUI screens, we enter the following claims, where P represents the reactor's control system:

  $$\Pi(P \sqsupseteq CoreTemp|FireWall) = 0.98.$$
  $$\Pi(P \sqsupseteq CoreTemp|(\neg FireWall \wedge ITDetection)) = 0.95.$$
  $$\Pi(P \sqsupseteq CoreTemp|(\neg FireWall \wedge \neg ITDetection)) = 0.93.$$
  $$\Pi(P \sqsupseteq ExtRad|FireWall) = 0.95.$$
  $$\Pi(P \sqsupseteq ExtRad|\neg FireWall) = 0.90.$$

- *Domain knowledge*. We submit the following domain knowledge under the form of predicates, where *indep(p,q)* means that events p and q are independent; one could questions whether some of the claims of independence are well founded, but we make these assumptions for the sake of simplicity.

  $$indep(FireWall, ITDetection).$$
  $$indep(P \sqsupseteq CoreTemp, P \sqsupseteq ExtRad).$$
  $$P \sqsupseteq SafeOp \Leftrightarrow (P \sqsupseteq CoreTemp \wedge P \sqsupseteq ExtRad).$$

– *Query*. We submit the query whether the claim

$$\Pi(P \sqsupseteq SafeOp|A) \geq 0.90,$$

is valid, where A is the assumption that the probability of *FireWall* is 0.90 and the probability of *ITDetection* is 0.80.

The system generates a theorem and submits it to Otter; it then analyzes the output file to determine if a proof was produced. The claim is deemed to be valid.

Whereas theorem provers are adequate for symbolic manipulations, what we need in our type of application is a combination of symbolic manipulation and numeric calculations. We have resolved this matter in this simple case by running two parallel inference threads, in a way, by declaring arithmetic operations to be evaluable (rather than simply symbolic), and adding clauses such as

```
((x+y)=z) <-> sum(x,y,z)).
((y=(z-x)) <-> sum(x,y,z)).
(sum(y,x,z) <-> sum(x,y,z)).
```

(where we deleted the quantifiers `all x y z`) to support symbolic equation manipulations and simplifications. In the long run, we may choose a different theorem prover, or a different means to infer queries from claims than theorem provers altogether.

5 Conclusion, assessment and prospects

In this paper, we have attempted to model system security (which we equate with survivability) on the basis of the following premises:

– First, we model security as a dependability property, alongside reliability and safety.
– Second, we model security using a refinement calculus, which has been used in the past to model reliability and safety.
– Third, we acknowledge the rigidity of strictly logical modeling, and derive a representation that supports probabilistic claims of correctness.
– Fourth, we integrate logical/probabilistic claims with cost functions, which allow us to quantify verification costs and failure costs.

We characterize the form of a security requirements specification, as well as the condition under which a system satisfies such requirements. The same reasons that preclude us from defining reliability as the absence of faults, also preclude us from defining security as the absence of vulnerabilities. Our definition of security (survivability) does not exclude vulnerabilities, and makes provisions for the cases when mishaps do not cause failure, or are otherwise recovered from.

Finally, we discuss means to support the management of security/dependability using the proposed models. Future prospects include the exploration of other forms of security (other than survivability), as well as the development and experimentation of the prototype.

References

1. Adams EN (1984) Optimizing preventive service of software products. IBM J Res Dev 28(1):2–14
2. Back RJ, von Wright J (1998) Refinement calculus: a systematic introduction. Graduate texts in computer science. Springer, Berlin Heidelberg New York
3. Becker SA, Whittaker JA (1997) Cleanroom software engineering practice. IDEA, Romania
4. Ben Ayed R, Mili A, Cukic B, Xia T (2000) Combining fault avoidance, fault removal and fault tolerance: an integrated model. In: Proceedings, design for safety workshop, NASA Ames Research Center, Moffett Field, CA, October 2000.
5. Ben Ayed R, Mili A, Sheldon F, Shereshevsky M (2005) An integrated approach to dependability management. In: Foundations of empirical software engineering: the legacy of Victor R. Basili, St. Louis, MO, Invited talk
6. Boudriga N, Elloumi F, Mili A (1992) The lattice of specifications: applications to a specification methodology. Formal Asp Comput 4:544–571
7. Boudriga N, Zalila R, Mili A (1992) A relational model for the specification of data types. Comput Lang 17(2):101–131
8. Boudriga N, Zalila R, Mili A (1993) Didon: a system for executable specifications. Inf Softw Technol 33(7):489–498
9. Desharnais J, Mili A, Nguyen TT (1997) Refinement and demonic semantics. In: Brink Ch, Kahl W, Schmidt G (eds.) Relational methods in computer science, Chap. 11, pp. 166–183. Springer, Berlin Heidelberg New York
10. Dyer M (1992) The cleanroom approach to quality software development. Wiley, New York
11. Ellison RJ, Fisher DA, Linger RC, Lipson HF, Longstaff T, Mead NR (1997) Survivable network systems: an emerging discipline. Technical Report CMU/SEI-97-TR-013, CMU Software Engineering Institute, November 1997
12. Gardiner P, Morgan CC (1991) Data refinement of predicate transformers. Theor Comput Sci 87:143–162
13. Hehner ECR (1993) A practical theory of programming. Springer, Berlin Heidelberg New York
14. Josephs MB (1987) An introduction to the theory of specification and refinement. Technical Report RC 12993, IBM Corporation
15. Linger RC (1993) Cleanroom software engineering for zero-defect software. In: Proceedings of the 15th Hawaii international conference on software engineering, Baltimore, MD, May 1993
16. Linger RC (1994) Cleanroom process model. IEEE Softw 11(2):50–58
17. Linger RC, Hausler PA (1992) Cleanroom software engineering. In: Proceedings of the 25th Hawaii international conference on system sciences, Kauai, Hawaii, January 1992
18. McCune W (1994) Otter3.0 reference manual and guide. Mathematics and Computer Science Division, ARGONE National Laboratory, January 1994.
19. McCune W (2003) Otter 3.3 reference manual. Technical Report Technical Memorandum No. 263, Argonne National Laboratory, Chicago, August 2003
20. Mili A, Cukic B, Xia T, Ben Ayed R (1999) Combining fault avoidance, fault removal and fault tolerance: An integrated model. In: Proceedings of the 14th IEEE international conference on automated software engineering, pp. 137–146, Cocoa Beach, FL, October 1999. IEEE Computer Society, Washington
21. Mili A, Sheldon F, Mili F, Desharnais J (2005) Recoverability preservation: a measure of last resort. Innov Syst Softw Eng A NASA J
22. Mills HD, Dyer M et al (1987) Cleanroom software engineering. IEEE Softw 4(5):19–25
23. Mills HD, Linger RC, Hevner, AR (1985) Principles of information systems analysis and design. Academic, New York
24. Nicol DM, Sanders WH, Trivedi KS (2004) Model based evaluation: from dependability to security. IEEE Trans Dependable Comput 1(1):48–65

25. Prowell SJ, Trammell CJ, Linger RC, Poore JH (1999) Cleanroom software engineering: technology and process. SEI series in software engineering. Addison Wesley, Reading
26. Shereshevsky M, Ayed RB, Mili A (2005) An integrated approach to security management. In: Cyber security and information infrastructure research group and information operations center, First Annual Workshop, Oak Ridge
27. Sommerville I (2004) Software engineering. 7th edn. Addison Wesley, Reading
28. Stevens F, Courtney T, Singh S, Agbaria A, Meyer JF, Sanders WH, Pal P (2004) Model based validation of an intrusion tolerant information system. In: Proceedings SRDS, pp. 184–194
29. Von Wright J (1990) A lattice theoretical basis for program refinement. Technical report, Department of Computer Science, Åbo Akademi, Finland,
30. Wos L (1996) The automation of reasoning: an experimenter's notebook with otter tutorial. Academic, Englewood Cliffs