A Practical Approach to Security Assessment

Darrell M. Kienzle *
The MITRE Corporation
kienzle@mitre.org

William A. Wulf
Department of Computer Science
University of Virginia
wulf@cs.virginia.edu

Abstract

Conventional approaches to building and assessing security-critical software are based on the implicit assumption that security is the single most important concern and can be the primary factor driving the software development process. Changes in the marketplace and the nature of security requirements have brought this assumption into question. There is now a large class of systems in which security must compete with other development goals. A risk-driven process model of software development provides a framework for building software that balances conflicting requirements. But a risk-driven process invalidates many of the assumptions made by conventional approaches to the specification and verification of security requirements.

This paper presents a new approach to assessing the degree to which software meets its security requirements. It does not propose a new specification notation or analysis technique, but provides a general framework into which existing notations and techniques can be integrated. It allows varying degrees of formality to be used: both across the components of the system, and through the development process. The appropriate degree of formality is whatever degree proves necessary to satisfy the stakeholders in the system that the security goals have been met.

This approach has been found to be theoretically appealing as well as useful in practice. Here we give a brief overview of the approach, explain how it integrates into a risk-driven process model, and discuss our early results in using it to assess, and thereby thus guide the development of, the Legion security model.

* Work performed while the author was with the University of Virginia Department of Computer Science and bears no past or present relationship to MITRE.

1. The Problem

The face of security is changing. In the past, systems were often grouped into two broad categories: those that placed security above all other requirements, and those for which security was not a significant concern. But such an all-or-nothing approach to security is getting increasingly harder to justify. Pressures to compete with commercial software have forced even the builders of the most security-critical systems to consider security as only one of the many goals that they must achieve. Meanwhile, market pressures and increased vulnerability from network attacks have forced commercial software vendors to recognize security as a major goal in their development efforts. There will certainly remain systems for which security is either of no concern or the single critical goal. But for an increasingly large number of applications, security must be balanced against other design goals.

Most existing research in the engineering of secure software has used formal methods in the context of a straightforward waterfall model of software development. Existing formal methods appear to work reasonably well in a conventional process model [Cor89, Kem90]. But, as Boehm points out, a waterfall process works well for systems where requirements and design issues are well understood from the outset [Boe88]. In the past, many security-critical systems exhibited these characteristics. In that environment, the waterfall model and conventional formal methods were generally adequate. However, they are much less useful in an environment where security and other design goals are in conflict.

When security must compete with other goals, the possibility exists that exploration of the design space might result in altering of requirements. Furthermore, the ability to verify security is only one of the many goals that the designer must consider. In such an environment, a risk-driven process model, such as the spiral model, offers significant advantages:

- The security ramifications of different design alternatives can be explored before the decision is made to commit to any one.
- A basic verification strategy can be laid out early in the process in order to avoid the unpleasant possibility that a workable design is impossible to verify.

- Decisions to relax security in order to meet other goals are done by design early in the process and not discovered by accident much later.

The conventional approach to formal specification requires that a single formal notation, exhibiting a single level of formality, be uniformly applied to the entire system. Yet any notation will be good at reducing certain risks but inappropriate for others. As Boehm notes, a risk-driven approach should allow greater formality to be applied to areas with greater risk and less formal notations to be used in areas with little risk. Similarly, design areas that pose the greatest risks should be explored at a greater level of detail earlier in the process [Boe88]. When security is important, the aspects that pose the greatest risk to the verifiability of system security need to be explored early in the process, when changes are most possible and least expensive.

This is consistent with Rushby's assertion that formal methods should be applied selectively to "the hard problems" [Rus95]. A risk-driven approach permits formal methods to be made more cost-effective, by only expending the costs in the area where the expected returns are greatest. It is true that such an approach is theoretically inferior to the total application of formality – areas that might have benefited from the application of formality may be overlooked. But a total application of formality to even reasonably-sized systems has proven to be practically infeasible [CGR95]. Selective application of formality is the only practical way in which the appropriate formal techniques can be applied where they can be of most use [Rus95].

It is unrealistic to believe that any system can be completely secure. Even if a completely secure system was theoretically possible, the very nature of verification limits the confidence that we can have in it [DLP79, Fet88]. There will always be something we can do to increase our confidence in the security of the system. For this reason, the driving question in this work has not been "is the system secure?" but rather "what should we do next to make the system more secure?"

In an environment where resources are limited, the spiral model is superior to the waterfall. If verification is left until after coding is complete, it may well never take place. While perfect verification may be theoretically appealing it is not attainable in practice. We should strive for the best verification possible, but at the same time recognize that there are limits to what we will achieve. In light of this, verification resources must be applied in as cost-effective a manner as possible. Resources should be applied to the areas where the greatest expected returns lie before they are applied to areas where they are expected to be less effective. Sometimes this means a concentrated application of formality; other times a broad application of less formal methods. Whatever the case, when verification resources are exhausted, the developer can be confident that the system was made as secure as possible given the development constraints.

2. Methodically Organized Argument Trees

This paper describes a new approach to security assessment. It is based on the presentation of security arguments using hierarchically organized trees, similar to the *fault trees* used by systems engineers to analyze the safety of critical systems. Each Methodically Organized Argument Tree (MOAT) encapsulates an argument that the system exhibits some desired security property. The different elements of the argument – the desired property, formal proofs, less formal reasoning, assumptions, axioms, lemmas, and component proof obligations – are all contained within the MOAT.

A simple example MOAT is presented in Figure 1. The example demonstrates how an argument regarding the privacy of e-mail in a given system might progress. This example is not intended to be complete or in any way convincing. It is presented only as an example of how a graphical tree structure can aid in organizing and presenting security arguments for further scrutiny.

MOATs permit the verification strategies for necessary security properties to be laid out in a clear, flexible, and accessible manner. They are claimed to:

- *Aid in communication between developers.* The exact relationship between system security goals and individual component requirements is documented. The key elements of a security verification strategy are identified. And a

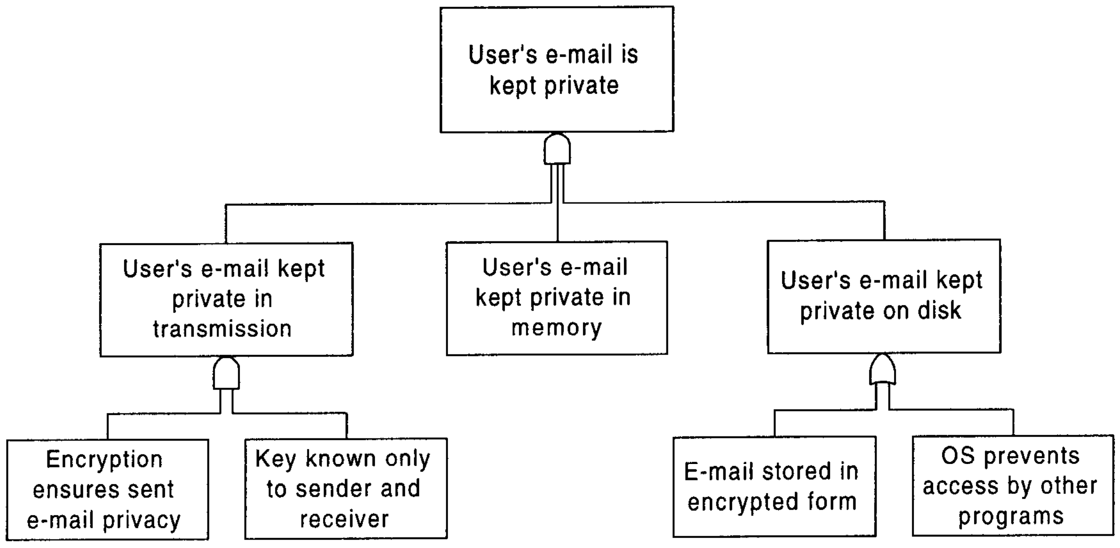
User's e-mail is kept private
User's e-mail kept private in transmission
User's e-mail kept private in memory
User's e-mail kept private on disk
Encryption ensures sent e-mail privacy
Key known only to sender and receiver
E-mail stored in encrypted form
OS prevents access by other programs

Figure 1 - A Simple Example MOAT

security design rationale is provided for later maintenance engineers.

- *Facilitate inspection.* The essential elements of security arguments are presented in a manner that makes them accessible and makes faulty reasoning apparent. This aids the independent validation process. Because of the open-ended nature of the analysis, additional resources can be expended in those areas where additional confidence is required.

- *Facilitate reuse of designs and their accompanying security arguments.* The approach makes the assumptions upon which an analysis depends explicit. Existing analyses can be reused as long as their assumptions can be justified – using the same process. More general knowledge and experience can also be reused, in the form of canned arguments and heuristics.

This paper presents initial results in attempting to validate these claims.

Readers who are familiar with system fault tree analysis will immediately recognize similarities between MOATs and fault trees. These similarities are intentional. But while the resulting trees may appear similar, the methodology used to construct MOATs is quite different from the traditional uses of fault tree analysis

The MOAT methodology has been found to permit the integration of existing security assurance techniques (including formal methods techniques) into a risk-driven process model. MOATs are developed concurrently with system specification, design, and verification. They permit high-risk areas to be more fully explored and more precisely specified before their lower risk counterparts. In particular:

- Different formal and informal notations can be used on different aspects of the system according to the nature and severity of the risks present.

- Different degrees of formality can be used as the project progresses. When the risk of changing specifications is great, an informal proof sketch can be created at the outset. This can then be refined into more formal versions as the specifications stabilize.

- The security ramifications of different design alternatives can be compared early in the process before expensive backtracking is required.

- A general verification strategy can be outlined and the tradeoffs between security verifiability and other properties can be explored early in the process.

- The greatest system risks – security and otherwise – are considered before less important risks. In a realistic development process in which resources are likely to be exhausted, this is critical.

Note that we are not proposing any new notation or analysis technique. The MOAT framework is intended to accommodate existing notations and techniques. We are also not advocating any particular existing notation or degree of formality. This approach is useful regardless of the degree of formality or the notations used.

3. Overview of the Method

The MOAT methodology involves building an argument tree (MOAT) for each security property that is to be analyzed. The process is iterative, continually refining the tree so as to make the argument that it presents more compelling. Tree construction loosely follows this sequence of steps:

```
Create a root node representing the desired
security property

Repeat
    Select the node representing the greatest
    remaining risk

    Replace its justification with a more
    compelling version

    If the justification is a decomposition
        Add the new child nodes

    If the property must be relaxed
        Propagate the change upwards

Until the risks have been sufficiently
addressed, or resources are exhausted
```

Each of these steps is described in greater detail in the sections that follow.

3.1 Initialization

The process begins with a goal – a security property that the system under construction is intended or required to have. Initially, a single node is created that simply asserts that the desired property holds – this assertion will be replaced with more compelling arguments as the tree is refined. This goal statement can be written in any notation, formal or otherwise. This goal is made the root of a tree structure. As the only node in the tree, the root is the only candidate for refinement in the first iteration of the body of the algorithm.

3.2 Justification

Initially, any leaf node represents an unjustified assertion. Justification involves associating with that node some argument as to why the asserted property holds. This can take many different forms. Some possibilities are:

- The property can be a system property that is decomposed into component properties. The justification would then represent an argument or proof that the component properties compose to provide the required system property.

- The property may be an assumption about the environment in which a system is to be used. In this case, the fact that it is an explicit assumption may be justification enough. Alternatively, justification could be presented as to why the assumption is valid for the system or its environment.

- The property might represent some derived design property. The argument could be a proof that a proposed design exhibits the property.

- The node might represent a property that an externally provided component must possess. The argument might then be that the creator has guaranteed this property. Alternatively, the argument could document the testing strategy and history of tests applied to the component.

- When multiple different notations are used in a single analysis, a node may represent the translation of the property expressed by its one child from one semantic basis to another. The justification for this translation might be a mechanical translation.

Whatever the format of the justification, additional assurance can always be provided. In particular, a justification that has been inspected and signed off on by many different parties may be considered more convincing than one seen only by the original author.

There is no single fixed degree of formality for these justifications. Some examples include:

- A journal-quality proof, where sufficient formality is used to convince the intended audience, but no more.
- A convincing argument that does not rely on any formal notation.
- A mechanically checked formal demonstration that source code meets its proof obligations.
- An informal argument that all known sources of failure have been accounted for.

Different nodes in the same tree may well contain different degrees of formality. This may be necessary due to different subject matter. For example, while it is possible to be very formal about source code properties, it might be very difficult to be formal about the fallibility of the users of the system. The former may involve mechanically checked proofs, whereas the latter may only contain an argument regarding employee training. Since security is a *system* property, MOATs have to accommodate all elements of a system, regardless of how amenable to the application of formality they might be.

3.3 The Order of Analysis

At each iteration of the process (other than the very first), the engineer is faced with the problem of deciding whether to decompose a leaf node or whether to refine an existing node. In addition, he or she must decide *which* node to perform these operations on.

The answer to both of these questions is that the engineer must decide where the greatest risk to security lies, and how to best address it. In some cases, this will mean a breadth-first approach in which system security properties are divided into the responsibilities of many different objects, in order to determine whether a "weak link" exists before committing more resources to the precise analysis of any one component. In other cases, a depth-first search might be required, constructing an argument that provides confidence that a single component will be able to meet its stated requirements before a system-wide solution that depends on that component property can be designed. In some cases, formality might be applied immediately; in others a rough proof sketch might suffice as evidence that a more formal proof will be possible at a later point in the process.

3.4 Decomposition into Subgoals

Many arguments will involve decomposition of the goal property into more manageable subgoals. In such cases, an AND-node is used to denote that all of the subgoals of the argument must hold true in order for the goal to hold. For example, if the goal property for a Kerberos-like system is to ensure that only authorized requests are executed, a decomposition might require that:

- The certificate authority will only issue certificates that permit authorized requests
- Requests will only be executed if permitted by a certificates issued by the certificate authority.

The AND node will contain the argument that these two subgoals do in fact compose to form the desired goal. The child nodes will each represent one of the sub goals, and will have to be justified in turn. Decomposition into subgoals has a number of typical uses:

Assumptions. A proof (or less formal argument) that relies on assumptions can be considered valid independent of its assumptions. However, the proven property can only be considered justified insofar as its assumptions can be justified. Thus, an AND node can be used to replace a goal property with an argument and subgoals representing the assumptions upon which the argument relies. In this manner, assumptions are made explicit rather than buried somewhere within the argument. Furthermore, because the process works from goal towards assumptions, the same process can be applied in an iterative manner in order to establish the assumptions.

In the example above, it might be assumed that the target objects are able to distinguish those certificates issued by the certificate authority. This assumption makes arguments regarding ticket existence easier, but needs to be justified at a later point in order for the dependant arguments to be considered applicable.

Component Requirements. A system property can be decomposed into individual component requirements. In the above example, the certificate authority would be responsible for ensuring that only authorized requests are given certificates. The protected services would be responsible for ensuring that only requests permitted by a certificate are executed. A user's confidence in the goal property will depend on (a) how accurate they believe the arguments supporting this decomposition to be and (b) how confident they are that the components meet their individual requirements. The latter can be achieved by applying multiple iterations of the MOAT analysis approach, or by using some other assessment technique.

Lemmas. Complicated mathematical proofs are often structured as a series of lemmas. Each lemma represents a manageable part of the proof. Each is established individually, and together they make the proof of the desired property far simpler. MOATs accommodate this approach by creating an AND node containing the final proof, with child nodes for each of the lemmas upon which it depends.

3.5 Decomposition into Alternatives

The other major form of decomposition is the use of OR-nodes to represent alternative arguments. For example, if the goal property is to ensure that a message is transmitted privately, the following decomposition could be used:

- The message is sent over a secure network.
- The message is securely encrypted.

From an entirely formal perspective, OR nodes are never really necessary. But they have several practical uses:

Reusability. OR nodes can be used to represent design alternatives. In the example above, the alternative approaches to demonstrating message privacy rely on different assumptions. An engineer reusing such an argument would be able to choose the alternative that better matched the assumptions of the target environment. Such flexibility makes the resulting arguments more suitable for reuse.

Flexibility. In the above example, the decision of whether to encrypt or rely on a secure network does not have to be made globally. Some objects may use one approach consistently; others may vary according to the nature of the communication. Such alternatives give the designers flexibility. For example, a given object might use encryption when communicating over insecure global channels, but get greater performance by sending messages in the clear over secure local networks.

Defense in depth. Engineering is a human endeavor and engineers are fallible. No matter how well a system (and its supporting arguments) is constructed, it can still fail. Defense in depth calls for a more robust system that accommodates failure - both of the individual components and of the design itself. In the message security example, this might involve encryption of messages sent over secure channels. If either one of these provisions fail, the other can still support the necessary property. While formal proofs may appear infallible, they are often based on informal assumptions. Physical security is not something that can be formally proven. Encryption is based on arguments regarding the secrecy of keys, which in turn are based on many different assumptions that might prove faulty. Even in mathematics, we often see multiple proofs of the same theorem—because proofs, like MOATs, are not intended to "prove" but rather to convince the reader [DLP79], and several arguments may be more convincing than any one.

3.6 Refinement

Refinement permits the engineer to revisit a node and replace the argument at that node. Typically this would be used to replace an informal justification with a more formal one. In this manner, an informal proof argument can be sketched out and then refined only after there is cause to believe that the general structure is sound and applicable to the design at hand. Some risks that arise from changing specifications can be ameliorated by letting the designer put off the expensive process of proof refinement until the specification has stabilized.

If a node property stated in an informal notation is replaced with more formal equivalents, this will possibly impact other nodes around the one in question. The child nodes may themselves have to be translated into the more formal notation. Conversely, the parent of the node being refined may now be refined as well, taking advantage of the greater formality in the statement of its child's property.

There is also the possibility that in refining the argument at a node, a problem with the less formal version will be uncovered. While the purpose of the informal argument sketch is to ensure that this will not cause major difficulties, this cannot be guaranteed. In the event that a problem arises, there are a number of possible ways of dealing with this:

- The argument may require additional assumptions, or other subgoals. In this case additional child nodes can be added.
- The argument may result in a weaker claim than was previously thought possible. It may be possible to address this via backtracking.
- The argument may be so flawed as to require replacement of the entire subtree rooted at that node.

3.7 Backtracking

Backtracking occurs when no argument can be constructed to justify the property required of a node. This can occur when a leaf node is initially being considered, or when a node is being revisited. In such an instance it is necessary to revisit the node's parent and perform one of the following:

- Determine an alternative decomposition that does not rely on the property that cannot be justified. Replace the subtree rooted at the parent with a new tree, perhaps reusing certain fragments of the subtree being replaced.
- Add an additional child node to the parent, thereby reducing the burden of proof on the node in question.
- Amend the property of the parent node so as to reflect a weaker property that can be established using a weaker property at the node in question. If this invalidates or weakens the arguments in the parent node's parent, the backtracking process continues with the parent node as the new node in question.

The backtracking process recognizes that there exists some risk that verification cannot be achieved as a rational process. The ease with which a design can have its security properties verified is only one of the many criteria by which a design can be judged. In performing this process, the design should attempt to balance verifiability with other concerns. No matter how much this goal is pursued, there will be cases where the process must backtrack.

This backtracking also recognizes that the stated security goals for a system may not be absolute. The stated goals are precisely that: goals. In performing this process, the engineer will uncover to what degree the stated security goals constrain the available design space. When conflict arises between the desired security goals and other design properties, the engineer will have to decide which has precedence. Sometimes, security will win out and the design will have to be modified to permit the stated goal to be achieved. But sometimes the security goals may have to be relaxed in order to accommodate other goals. The inclusion of backtracking in the process provides for both of these alternatives.

This is particularly important in a reuse-centered environment. If every system is to be built from scratch, it is not unrealistic to require that a very specific set of security goals be achieved. But when standard components are being used, the designed will have to decide whether to adapt the definition of security to match what can be provided using existing part, or whether the need for a specific definition of security warrants the creation of novel components. The scope of this reuse can range from individual functions in a source code library to complete commercial off-the-shelf software products. The process recognizes the need to adapt the definition of security to what is viable given the other design parameters. But it also ensures that the user is *explicit* about any alterations of the stated security goals.

3.8 Termination Criteria

This repetitive decomposition and refinement process can in theory continue indefinitely. In practice, it will be necessary to stop at some point. Depending on the manner in which this technique is used, there can be vastly different termination criteria:

- The analysis can be terminated when the engineers believe that no sufficiently large threats to security remain. Of course an outside validation agency might disagree and require that it be resumed from the point at which it was left off.
- When the deadline for product deployment is reached, the analysis can be halted. Alternatively, it can continue throughout the lifecycle of the product and only halt when the product is finally retired from active use.
- If a single formal notation is being used, the analysis can stop when no more analysis can occur because the basic assumptions have been reached or a uniform degree of formality in the reasoning process is achieved.
- When resources allocated to the enforcement of security have been exhausted, the analysis process can halt.

Because of the open-ended nature of this approach, the analysis may never be "complete" in any meaningful sense. This is by design. Additional resources can always be added to the analysis, and the analysis will always apply the resources available thus far to the greatest security risks. Ultimately, the decision when to terminate the process will lie in the hands of the engineer.

3.9 Assessment

There are several ways in which MOATs can be assessed. In the simplest, every leaf can be considered to be either true or false. Using the Boolean relations described at the interior nodes, the goal property can be determined to be either true or false. But such an absolute argument will depend on the assumption that the leaves are labeled correctly, and that the arguments at the interior nodes are correct.

Alternatively, if a user of this method can assign meaningful quantitative values to the leaves, and establish meaningful ways of combining these values at the interior nodes, conventional fault tree solving methods can be used to compute quantitative values for the goal property. Such quantification is not presently possible, but it is not precluded by the use of MOATs.

Perhaps the most useful approach to assessment is to simply examine the arguments at the interior nodes and the justifications presented for the leaf nodes and decide whether they are convincing. Some users may require formal proofs. Others may be more concerned with the arguments that justify the assumptions. In any case, the method accommodates whatever technique a given user may require.

The assessment process is actually quite simple. The various stakeholders simply study the analysis and decide whether or not they are satisfied that the analysis (a) does in fact reflect the actual system and (b) demonstrates that the required security property is enforced. While assessment that depends on human judgment is quite informal and contains the potential for abuse, there are good reasons why this approach may be superior to the formal approach:

- It can be used at all phases of the process; not just when completely formal proofs have been constructed.
- It permits a risk-based approach to be taken, where correspondingly greater efforts are made to address the greater risks.
- It admits probabilistic arguments and arguments based on effects that are difficult to quantify, such as deterrence. Many defense mechanisms are not absolutely secure, but are considered "strong enough."
- It recognizes that no proof is completely formal and that there is risk in even the smallest element of informality.
- Proofs must be convincing. Formal approaches that are not mechanically checked can hide flawed reasoning behind inaccessible notations.

4. Analysis of the Method

The MOAT approach represents a considerable change from conventional approaches to security assessment. As such, resistance is to be expected. In this section we consider some of the more troublesome aspects of such a paradigm shift and attempt to assuage them.

4.1 Imperfect Methods

One of the most common concerns voiced about utilizing different degrees of formality is that such an approach may well miss something that a uniform application of formality would have detected. Much of the effectiveness of formal methods arises from the fact that they leave no stone unturned. The cost of this effectiveness is turning over a lot of stones under which nothing interesting is found.

This is a valid concern. If we knew a priori where all the ambiguities in an informal specification were, they would not pose a problem. Formal specifications are effective in part because they identify areas of ambiguity that were not previously known to be ambiguous. Formal proofs are effective because they highlight reasoning flaws that might otherwise have gone undiscovered. From a theoretical standpoint, the selective application of formality will always be inferior to a complete application of that formality.

But this is a bit of a strawman argument. If the costs of complete application of formal methods preclude their use, then selective application of those same methods is the only practical alternative. Arguments are often heard that practitioners *should* be using formal methods and that the factors preventing their use in practice are *myths* [Hal90, BH94]. But the fact remains that the mainstream software engineering community has largely failed to embrace a complete switch to formal methods. Even formal methods proponents have begun to recognize that the selective application of those formal methods may be the only viable alternative [Rus95, CGR95].

Furthermore, the MOAT approach does not preclude a completely formal approach. If the resources are available, a completely formal approach can be taken. The MOAT approach does attempt to order the application of these resources, so that:

- Backtracking is reduced.
- Formality is applied where it is most cost-effective, before being applied to less cost-effective areas.
- Specification, design, and verification can occur concurrently.

Such an approach provides insurance against the possibility that resources are exhausted before complete application of formality is achieved. It is much like the "inverted pyramid" that journalists follow, presenting the elements of a story in order of importance. This permits an editor to cut the story at any point and ensure that the space provided has been used in an optimal manner. Under the MOAT approach, if security resources are exhausted, those available have been used in the most cost-effective manner possible.

4.2 The Potential for Abuse

Another concern about this approach is that it could potentially be abused. As the method permits the user to decide what degree of formality should be used, there is no mechanism to ensure that they will not choose a notation that permits them to sweep the more difficult details under the rug and claim to have analyzed the system sufficiently. This may not be malicious – a user of the method may simply not understand the degree of rigor that is appropriate to the security risks posed by the system. For these reasons, it might appear that a more rigorous process should be prescribed and the decision taken out of the designer's hands.

Although greater rigor in our processes is certainly a goal, there are many examples of techniques that are useful in spite (or perhaps even partly due to) their lack of rigor. Perhaps the most prominent example is fault tree analysis, used by systems engineers to reason about safety-critical systems. Fault trees are the single most important safety assessment technique. The Nuclear Regulatory Commission uses them to validate the safety of nuclear reactors. The military uses them to analyze the detonation systems of nuclear warheads. And Underwriters Laboratory uses them to analyze the safety of household products.

Yet in spite of their importance, their construction and validation depends on human insight and experience. One textbook puts it "construction of fault trees is an art as well as a science and comes only through experience" and then proceeds to present a list of general heuristics to guide in their construction [McC81]. Both rigorous processes and automated software exist to aid in the automatic construction of fault trees from system diagrams, but even they are based on insight and experience rather than formal reasoning [RM83, HK85]. They also require a model of the system that accurately reflects the component interactions and environmental assumptions, which in turn is typically validated using insight rather than logic.

Fault trees strike a careful balance between rigor and flexibility. They accommodate arbitrary designs and yet exploit reuse of experience with standard designs. They can be buttressed by formal reasoning about the accuracy of their construction, or they can depend on appealing to the intuition of the reader. It is up to the stakeholders in the system to determine what degree of rigor is required. It is up to them to ensure that the ability to accommodate different degrees of formality is not abused. That fault trees have become so popular is a sound argument that responsible practicing engineers can be entrusted to make these decisions wisely.

4.3 Generality

The MOATs approach is actually a generalization of existing formal and informal methods. It does not *dictate* a specific formal method and it does not *require* a single consistent formal semantics. But that does not stop a user of the method from opting to use a single existing notation with a single semantic basis. If a user wants to build a complete formal specification in Z, the MOATs approach can still provide benefits in that it allows the various proof sketches to be laid out before the cost of complete formalization is incurred. Furthermore, it makes the resulting proofs more accessible, by requiring the user to structure the analyses in a manner that makes the interdependence of component properties explicit.

The MOATs approach is not wedded to existing formal notations, however. It is critical to remember that we do not use formal methods for formality's sake alone. Rather, the use of any formal technique represents a risk-reduction activity. If we view formal methods as nothing more than risk-reduction techniques, it is simple to see how formal methods can be adapted to fit into a risk-driven process model. Consider the following examples of specification as simply a risk-reduction activity:

- Parnas's specification of the A7 control software showed how a notation can lack a formal semantics and yet be tailored to the risks that were anticipated. The input/output forms captured knowledge about the most common failure modes in the hardware / software interface. And the tables represented years of experience with specifying state-driven, process-control software [Hen80].
- The most successful Z specifications take advantage of the experience gained in using that language to develop large database systems. Many published specifications demonstrate the utility of the language and its reusable toolkit at specifying the high-risk element of these systems – internal database consistency – while essentially ignoring the low-risk – the sources of database events and their inputs and outputs [Hay85].
- Specification of GUI systems is performed largely using screen-painting utilities with which there are no formal semantics associated. Nevertheless, these approaches address the key risks of these systems – that they will not be intuitive or will not meet the customer's expectations.
- The most common specification tools in business are form and report generation utilities. These are not general-purpose formal specification languages, but they do have an (operationally-defined) formal semantics, and they address the exact risks that the users are concerned with.

All of these examples provide evidence that the important question is not whether a specification notation is formal enough, but whether it addresses the key risks.

On a different note, the MOATs approach also appears to be a generalization of some other work in the area of reuse of security verification. Frincke [Fri96] discusses the use of design templates with pre-verified security properties. These could be incorporated directly into the MOATs approach by creating a node that merely references those arguments and has children representing the assumptions upon which the reused arguments rely. Spencer [Spe96] discusses a checklist-approach to the determination of security requirements. This approach could be approximated in the MOAT approach by using a node with one child for each element in the checklist. Each child would then either contain a rationale as to why it does not apply, or a translation of that general checklist entry to the specifics of the system in question. Both of these approaches are claimed to be quite useful by their authors, and both are accommodated by the MOAT approach.

4.4 The Rational Process

The cost-effectiveness of the MOATs approach will depend on the user's ability to choose risks well. With perfect foresight however, a rational design process [PC86] can be achieved. With less than perfect foresight, the amount of backtracking required will depend on the skill of the user at anticipating risk areas. This is true of *every* development methodology – both those that explicitly acknowledge the role of risk in the development process and those that don't. The MOAT approach makes this role explicit in order to accommodate different development paths. Ultimately, however, the assessment arguments will either convince the intended audience or not. The route taken in developing the assessment arguments will not matter.

This is a critical point. Poor employment of the method will not permit insecure systems to be mistakenly judged secure. At worst, the verification resources will have been wasted. But the assessment arguments will either convince the stakeholders who are concerned with security, or they will not. Regardless of the actual process taken, the rational process can be approximated, and it will be the products of that process that will ultimately be assessed. The open-ended nature of the MOAT approach does have the significant advantage of being amenable to additional justification should the original analysis require strengthening.

Fraser, et al [FKV94] presented a survey of popular formal specification approaches. In the more general-purpose notations considered, formal notations were provided without methods for the elaboration of specifications written in those notations. Users of these notations are left completely to their own devices when it comes to actually creating a specification. The specifications might as well be pulled out of thin air, as far as the befuddled new user is concerned. The MOAT approach attempts to fill this void by giving users some guidance as to which parts of the specification contain the greatest risk and should be considered early on. Less critical aspects can always be elaborated at a later point – if at all. MOAT also aids users in helping them structure their arguments informally before attempting to commit them to a formal notation.

5. Empirical Evaluation

This approach grew out of our experience in developing the security model for the Legion distributed system [WWK96]. It was developed to address the need for organization and preservation of informal discussions about the security ramifications of design alternatives. As time progressed, the approach was modified in order to better meet our needs and to address its shortcomings.

We report here on some of our experience with using this approach in practice. Although we attempted to make these experiments as controlled as possible, there is still considerable room for experimental error. Some of the more significant caveats are:

- The approach was only used by a single group – one that included its developers.
- It was not compared to other assessment methods used on the same projects.
- The experiments were not numerous or diverse enough to draw general conclusions.
- The approach evolved during the experiments.

In spite of this, we found the approach to be quite practical. We present some of the more interesting observations here and leave it to the user to decide whether these results should be considered promising. We report here on three specific applications of MOATs to aspects of the Legion security model. These three experiments vary considerably in both scope, and formality.

5.1 The Legion Caching Mechanism

One of the key elements of the Legion security model is the MayI function, which is used to define discretionary security for an object. The MayI function issues licenses that can be cached in order to enhance performance by exploiting temporal locality in method invocations. The developers were concerned that the subtle interactions between cache and MayI might weaken user-defined security policies in non-obvious ways.

The application of the analysis process to the Legion caching mechanism produced the following general observations:

- Using the method we were able to determine the precise criteria for Legion system compliance. Originally we had intended to create a complete formal specification of the Legion model and then require that an implementation conform to the model in order to be considered compliant. Through this process we were able to trace the required security property to specific requirements on the Legion model. Rather than require compliance with all aspects of the formal model, we can discover and state the specific properties that must be met in order for a system to be considered compliant.
- The method also demonstrated both the need for and the utility of backtracking. By starting with a desired security goal and then relaxing it in order to accommodate design choices, we were able to precisely derive a reasonable compromise between security and performance. In this manner the method provided a valuable distinction between the security goals we would like to achieve and those that we can reasonably expect to achieve.
- The method was used at a fairly low level to analyze a specific design. We found the method useful at this level. We were able to extract specific code-level verification obligations that could be targeted during the implementation to follow. If it later proves necessary, these verification obligations can be used to rigorously verify the code.
- This analysis demonstrated to our satisfaction that much of the benefits of rigor could be achieved without incurring the high costs of formalization. While risks certainly exist at the

code level, our analysis provides confidence that the basic design is sound.

5.2 The Legion Delegation Model

The second significant analysis performed using this method was an analysis of the Legion delegation model. Most every multi-user system permits programs and users to act on behalf of another program – most typically to permit a single executable to have the rights of the user that invoked it. In a distributed system it can be very difficult to ensure that an object claiming to be working on behalf of another object really is authorized to perform the actions it attempts.

Figure 2 presents the skeleton of the argument that resulted after several iterations and review cycles. The RA refers to the Legion *Responsible Agent* – the object on whose behalf an object invoking a method on *target* claims to be working.

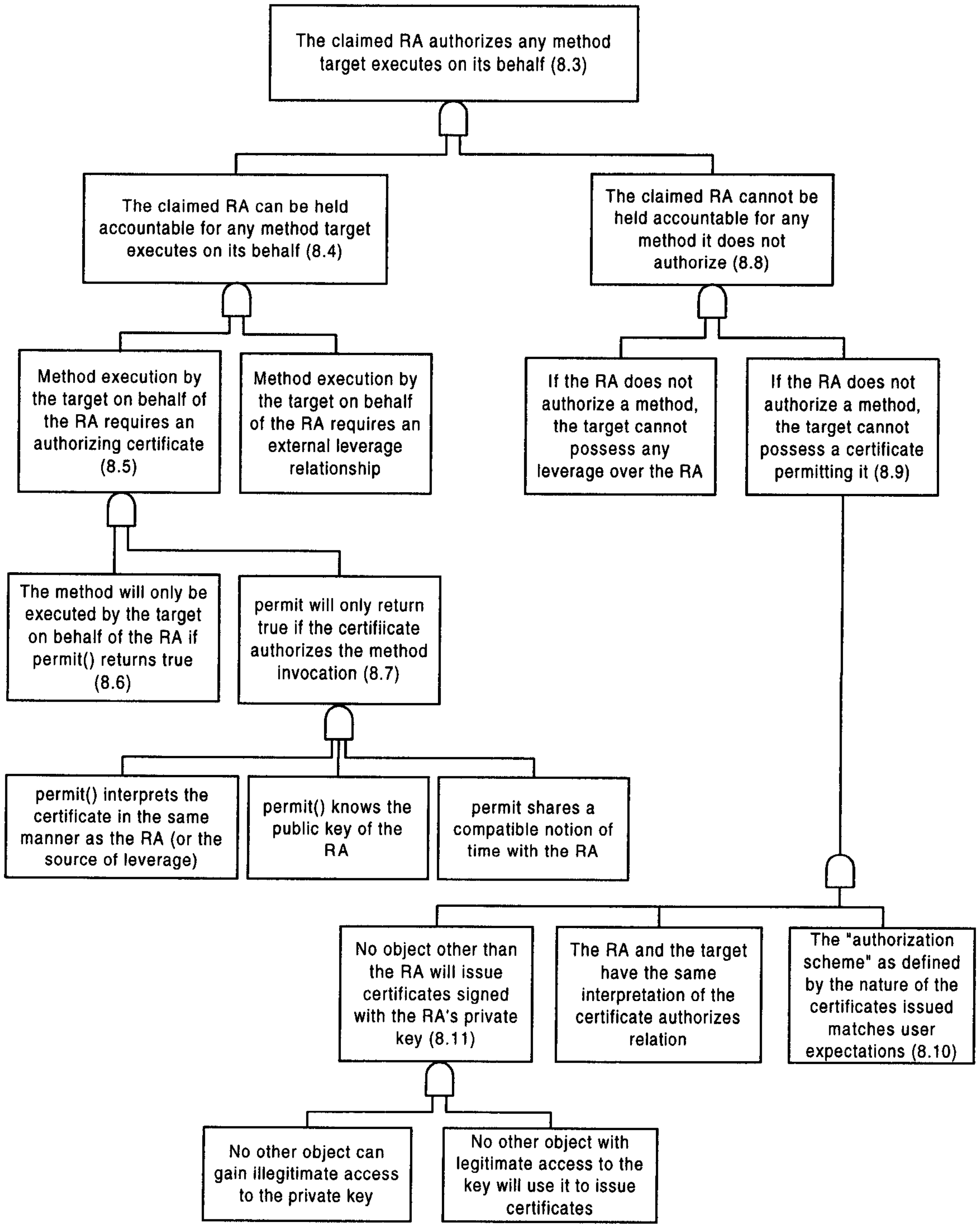
The claimed RA authorizes any method target executes on its behalf (8.3)
The claimed RA can be held accountable for any method target executes on its behalf (8.4)
The claimed RA cannot be held accountable for any method it does not authorize (8.8)
Method execution by the target on behalf of the RA requires an authorizing certificate (8.5)
Method execution by the target on behalf of the RA requires an external leverage relationship
If the RA does not authorize a method, the target cannot possess any leverage over the RA
If the RA does not authorize a method, the target cannot possess a certificate permitting it (8.9)
The method will only be executed by the target on behalf of the RA if permit() returns true (8.6)
permit will only return true if the certifiicate authorizes the method invocation (8.7)
permit() interprets the certificate in the same manner as the RA (or the source of leverage)
permit() knows the public key of the RA
permit shares a compatible notion of time with the RA
No object other than the RA will issue certificates signed with the RA's private key (8.11)
The RA and the target have the same interpretation of the certificate authorizes relation
The "authorization scheme" as defined by the nature of the certificates issued matches user expectations (8.10)
No other object can gain illegitimate access to the private key
No other object with legitimate access to the key will use it to issue certificates

Figure 2 – Skeleton of the Legion Delegation Analysis

The analysis process was applied to this problem in order to gain a greater understanding of the issues involved and explore the tradeoffs between security, performance, and flexibility. A general model was posited and then analyzed to determine whether it could hold up under the many demands placed on it, and whether obvious improvements appeared.

Unlike the analysis of the caching mechanism, which was limited to a single design, this analysis was intended to be as generic as possible, accommodating multiple design alternatives. The analysis of delegation also differed in that it included objects that could be placed on different hosts, and be created by different users with different goals. The following results were observed from this analysis

- The method aided in explicitly differentiating between an intuitive notion of delegation and an enforceable definition thereof. The user is given a clear understanding of the relationship between flexibility and enforceability.

- In performing the analysis, we discovered that we were able to replace vague, intuitive notions of "trust" with specific required properties. We had always believed that "objects have to trust their host". This vague requirement was replaced (for this property) with specific requirements that hosts not issue delegation certificates on behalf of the object and that they not permit access to the private key of the object. Replacing fuzzy notions of trust with specific requirements in need of assurance was a very useful result.

- The method revealed that certain implicit assumptions that are often made of closed systems are invalidated by the open Legion environment. In particular, while a given design may ensure that an object is "willing to take responsibility" for the actions it delegates, it cannot ensure that the object can actually be held responsible in any meaningful manner. The analysis identified the need for some form of accountability relationship between objects.

- The resulting analysis proved very accessible. One engineer who saw this analysis but was neither familiar with the system nor a specialist in security was able to see a very subtle difficulty with issues of time. Once the concern was raised it was easy to understand the problem. Nevertheless, this remained a clear validation of the accessibility of the approach.

- The analysis identified several general patterns (heuristics) that reappeared in later analyses regarding other topics. For example, the basic alternative of either preventing failure or detecting failure is something that can be applied in many contexts. Similarly generic is the notion of using a precisely defined protocol as an oracle in order to as a means of moderating between mutually suspicious objects created with opposing goals.

- The analysis only considered those properties that could be directly traced to the primary goal of enforcing delegation. One very difficult element of the problem – the problem of getting expired certificates refreshed – surprisingly was found to play no role in the analysis. On later reflection this made perfect sense. But it took the use of a rigorous process to isolate those elements that are crucial to security from those that are merely optimizations.

5.3 Legion Mandatory Security

The Legion security model provides mechanisms whereby users can define arbitrary discretionary security policies. These same mechanisms can also be used to implement mandatory security policies over collections of objects. However, these two different uses of the same mechanisms imply different properties that those mechanisms will have to exhibit. Initial implementations of the Legion system will only concentrate on discretionary security. But it is critical to convince potential users that the basic strategy for mandatory security is sound. An analysis was performed with the intent of convincing users that the model could accommodate mandatory security policies, albeit under later implementations.

The analysis of mandatory security was performed at a high level and in a very abstract manner. It was significantly different from the other analyses in that it was intended to demonstrate the feasibility of proofs under the proposed model. This analysis led to the following general observations regarding the method:

- The analysis allowed covert channels to be considered within a *system* context. In particular, the risk of computerized covert channels was balanced against the risk of sensitive data being leaked via conventional means. Given this context, it is hard to justify either restricting functionality or expending assurance resources against covert channels.

- The analysis of mandatory security demonstrated the utility of the method at accommodating a variety of degrees of assurance. Different assurance arguments were provided for the user to choose from. They ranged from the conventional "either it can be made completely secure or it should not be built" variety to less easily quantified arguments concerning the difficulty of penetrating a given site and the deterrent effects of threatened detection. The method proved flexible in accommodating both ends of this spectrum.

- Like the delegation analysis, this analysis demonstrated the utility of the method in replacing vague definitions of trust with specific component obligations

- The mandatory security analysis effectively uncovered certain elements of the design that will be key to future verification efforts. In particular, it was discovered that later analyses would depend heavily on certain ordering restrictions being placed on the Legion protocol stack. Armed with this knowledge, the designers can work around these verification needs without constraining the design space unnecessarily.

- The mandatory security policy made clear that the ability to enforce a mandatory policy will depend heavily on how that policy is defined. In particular, a policy defined in terms of raw message traffic is much easier to enforce than one defined in terms of method invocations, but is also far less expressive. The application of this method reveals these tradeoffs before requiring a lot of investment in formal notations.

- This analysis was initially quite informal. But the areas of greater risk were explored in greater detail. Here risk is considered to be the concerns of potential customers. We were able to construct the analysis in an iterative manner, focusing resources on the areas where customers were most skeptical. The open-ended nature of the analysis proved quite valuable, in that the appropriate degree of rigor could be converged on rather than having to be selected a priori.

6. Conclusions

From both our analytical consideration of the method and our initial empirical evaluation, we can draw some early conclusions. They are grouped into the following broad categories:

- *As a medium for communication.* MOATs have proven themselves very useful in this area. They isolate the essential arguments from all of the supporting notational concerns. This makes these arguments more accessible both to developers and to other stakeholders in the system. The precise documentation of the relationship between component properties and system-level security allows developers of these components to understand exactly their role in ensuring security and the ramifications of their decisions.

- *As part of a risk-based process model.* We have empirically observed MOATs to be useful at all phases of the software lifecycle. Unlike many other formal techniques, MOATs are flexible enough to adapt as the software requirements inevitably change. MOATs have even proven useful at uncovering potential conflicts in requirements, and thereby facilitating the negotiation of a reasonable compromise. Finally, the open-ended nature of the process means that the analysis can be constructed in a manner that is both responsive to feedback and ensures that verification resources are applied in the most cost-effective manner possible.

- *Reusability.* We have found the MOAT methodology to be amenable to the reuse of knowledge in the form of general heuristics. The notational flexibility has also made it easy to reuse more substantial analyses. And because of the explicit identification and separate justification of assumptions, reuse of parts of existing analyses has proven possible. We also believe that other work in the area of reuse of security analyses demonstrates the viability of this more general approach. Still, considerable more work has to be done in order to draw any stronger conclusions in this area.

- *As a vehicle for exploration.* Clearly, it is difficult to claim that the things we found using this approach wouldn't have been found using any other approach. It is difficult to separate the abilities of the users from the utility of the method. Some of the things we uncovered were clearly due to moments of inspiration. Nevertheless, we find this approach to be extremely promising. As with any formal method, the moments of insight occurred as a result of attempting to document formerly vague ideas. However, in contrast to other formal approaches that we have used, the MOAT approach encourages the user to find the degree of rigor that is most suitable to each aspect of the problem at each point in the design process. The straightjacket of conventional formal methods is thereby replaced with a much more comfortable alternative. And any approach that users find practical will be more effective than one that is left on the shelf because it isn't quite right and isn't flexible enough to be altered.

Acknowledgements

We are grateful to Worthy Martin, John Pfaltz, Kevin Sullivan, Carl Landwehr, and Cathy Meadows for insightful and constructive feedback on this work. This work was partially supported by DARPA (Navy) contract # N66001-96-C-8527 and DOE grant DE-FD02-96ER25290.

References

[BH94] Bowen, J., M. Hinchey, "Seven More Myths of Formal Methods," Oxford University Computing Lab, Technical Report PRG-TR-7-94, June 1994.

[Boe88] Boehm, B., "A Spiral Model of Software Development," IEEE Computer, May 1988, pp. 61-72.

[Bos95] Boswell, A., "Specification and Validation of a Security Policy Model," IEEE Transactions on Software Engineering, Vol. 21, No. 2, Feb. 1995, pp. 63-68.

[CGR95] Craigen, D., S. Gerhart, and T. Ralston, "Formal Methods Reality Check: Industrial Usage," IEEE Transactions on Software Engineering, Vol. 21, No. 2, Feb. 1995, pp. 90-98.

[Cor89] Cornwell, M., "A Software Engineering Approach to Designing Trustworthy Software," Proceedings of the IEEE Symposium on Security and Privacy, Oakland 1989, pp. 148-156.

[DLP79] DeMillo, R., R. Lipton, and A. Perlis, Social Processes and Proofs of Theorems and Programs, Communications of the ACM, Vol. 22, No. 5, May 1979, pp. 271-280.

[Fet88] Fetzer, J., "Program Verification: The Very Idea," Communications of the ACM, Vol. 31, No. 9, Sept. 1988.

[FKV94] Fraser, M., K. Kumar, and V. Vaishnavi, Strategies for Incorporating Formal Specifications in Software Development, Communications of the ACM, Vol. 37, No. 10, Oct. 1994, pp.74-86.

[Fri96] Frincke, D., Developing Secure Objects, Proceedings of the 19th National Information Systems Security Conference, Baltimore Maryland, October 1996, pp. 410-419.

[Hal90] Hall, A., "Seven Myths of Formal Methods," IEEE Software, Vol. 7, No. 5, Sept. 1990.

[Hay85] Hayes, I., "Applying Formal Specification to Software Development in Industry," IEEE Transactions on Software Engineering, Vol. SE-11, No. 2, Feb. 1985, pp. 169-178.

[Hen80] Heninger, K., "Specifying Software Requirements for Complex Systems: New Techniques and their Application," IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, Jan. 1980.

[HK85] Henley, E. J. and H. Kumamoto, *Designing for Reliability and Safety Control*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985, pp. 407 – 458.

[Kem90] Kemmerer, R. A., "Integrating Formal Methods into the Development Process," IEEE Software, Sept. 1990, pp. 37-50.

[McC81] McCormick, N. J., *Reliability and Risk Analysis*, Academic Press, San Diego, California, 1981, pp. 154-228.

[PC86] Parnas, D., P. Clements, "A Rational Design Process: How and Why to Fake It," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, Feb. 1986.

[RM83] Roland, H.E. and B. Moriarty, *System Safety Engineering and Management*, John Wiley and Sons, New York, 1983, pp. 215-271.

[Rus95] Rushby, J., "Formal Methods and their Role in the Certification of Critical Systems", SRI-CSL-95-01, January 1995. http://www.csl.sri.com/csl-95-1.html

[Spe96] Spencer, R., Deriving Security Requirements for Applications on Trusted Systems, Proceedings of the 19th National Information Systems Security Conference, Baltimore Maryland, October 1996, pp. 420-427.

[WWK96] Wulf, W. A., C. Wang, and D. M. Kienzle, "A New Model of Security for Distributed Systems", Proceedings of the New Paradigms in Security Workshop, Lake Arrowhead, California, 1996.