Fault trees for security system design and analysis

Phillip J. Brooke[a] *and Richard F. Paige*[b]

[a]*School of Computing, University of Plymouth, Drake Circus, Plymouth, PL4 8AA, UK; email: philb@soc.plym.ac.uk*

[b]*Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK; email: paige@cs.york.ac.uk*

ELSEVIER

Computers & Security
Vol 22, No 3, pp256-264, 2003

0167-4048/03

Abstract

The academic literature concerning fault tree analysis relates almost entirely to the design and development of safety-critical systems. This paper illustrates how similar techniques can be applied to the design and analysis of security-critical systems. The application of this technique is illustrated in an example inspired by a current public-key cryptosystem.

Keywords: Security. Fault tree analysis.

1 Introduction

Fault tree analysis (FTA) is a "method for acquiring information about a system" [1]. It is regularly referenced in the context of developing safety-critical, i.e. high-integrity software. The aim in these contexts is to analyse a system design to identify areas of risk which can then be addressed. There are a number of commercial tools [2] which support the method by providing drawing and probabilistic calculation facilities.

However, there is little in the academic literature on the application of FTA to security-critical systems; one of the few discussions is in Anderson's recent book [3]. Another is Helmer et al.'s description of the use of software fault trees specifically for intrusion detection systems [4]. This paper attempts, in part, to contribute to the understanding of how to apply FTA to security-critical systems.

The work in this paper came about in an attempt to apply FTA to a system being developed at the Communications-Electronics Security Group (CESG), UK. This system, while relatively small, had a significant amount of inherent complexity with dependencies between components, and was security-critical.

This paper briefly reviews the FTA technique in Section 2. We then consider related approaches to modelling secure systems and security properties in Section 3. We outline how FTA can be used for security in Section 4, and then illustrate this with an example in Section 5. The paper finishes with a discussion and conclusion (Sections 6 and 7 respectively).

2 Review of FTA

2.1 FTA concepts

Fault tree analysis is a deductive technique which focuses on one particular undesired event [1]. From that initial undesired event, a tree is constructed with that event at the root. From then, each event is either a primary event (i.e. an event that is not developed further in the analysis; described further later in Section 2.1.1), or an INTERMEDIATE event. INTERMEDIATE events are then developed by being connected to a gate, which is itself connected to other events.

Fault trees are classically associated with safety-critical systems, for example, systems associated with nuclear reactors or weapons, or aircraft flight systems. Hence, possible undesired events are, "The safety valve does not open", or "The aircraft crashes".

Events within a fault tree are typically discrete occurrences of a fault, or some event that while itself is not a fault, can contribute to a fault higher up the fault tree. The fault tree then qualitatively illustrates how the undesired event results from the primary events, via zero or more intermediate events.

The symbols we may encounter in a fault tree are given in Figures 1, 2 and 3. The gate symbols are clearly inspired by logic circuit notations: they should be familiar to most

readers. The symbols in the figures, and the descriptions that now follow, are drawn entirely from the Fault Tree Handbook [1].

2.1.1 Event symbols

The BASIC, UNDEVELOPED and EXTERNAL events are the *primary events*. They are not developed any further in the analysis for some reason. Typically, this is because we are not looking any more closely at that aspect of the system: it is outside the scope of the analysis.

- The BASIC event indicates a basic initiating event at the limit of resolution; i.e. we do not wish to develop this fault any further in this particular analysis.
- The UNDEVELOPED event is undeveloped because there we either lack information, or the event is of no consequence.
- The EXTERNAL event is an event that is expected to happen in the course of normal operation of the system.
- The INTERMEDIATE event is neither the top-most undesired event, nor a primary event. Further events are attached to the INTERMEDIATE event, usually via gates.
- The final event is the CONDITIONING event. It is used primarily with the INHIBIT and PRIORITY AND gates: it will be described shortly.

2.1.2 Gate symbols

The first three gates should be familiar to most readers in computing:

- The AND gate indicates that the output fault (drawn above the gate) only occurs if the two (or more) input faults (drawn below the gate) occur.
- The OR gate indicates that the output fault occurs if at least one of the two (or more) input faults occur.

Figure 1: Fault tree event symbols [1, Table IV-1].

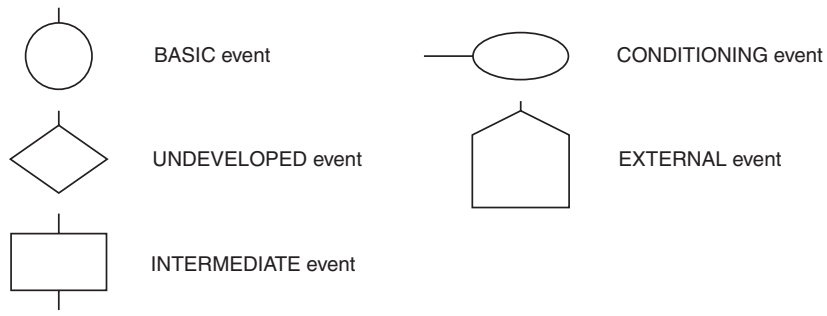
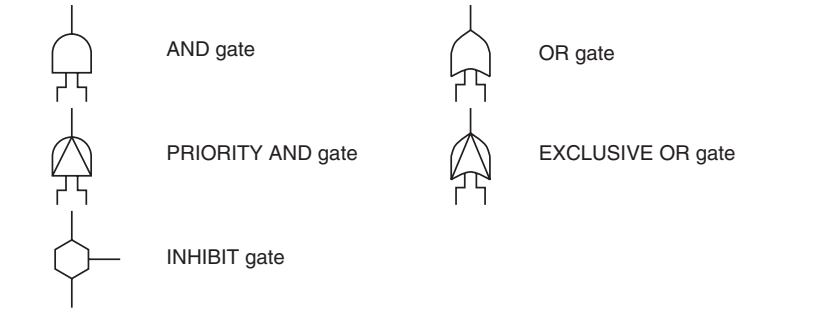


Figure 2: Fault tree gate symbols [1, Table IV-1].



- The EXCLUSIVE OR gate indicates that the output fault occurs if exactly one of the two (or more) input faults occur.

The PRIORITY AND gate indicates that the output fault only occurs if all the input faults occur in a specified order (left to right on the page). (An alternative notation for this in the Fault Tree Handbook [1] uses the normal AND gate and a CONDITIONING event to state the constraint.)

The INHIBIT gate is another special case of the AND gate: The output event is caused by a single input event, but a particular condition (given by a CONDITIONING event) must be satisfied before the input can produce the output. This condition may be probabilistic. The difference from the AND gate here is one of emphasis: the condition given is not a fault or external event as such, but something to do with the environment, e.g. a particular temperature.

Figure 3: Fault tree transfer symbols [1, Table IV-1]

Phillip Brooke

Phillip Brooke is a lecturer at the University of Plymouth, United Kingdom, where he works with the Network Research Group. He completed his DPhil in Computer Science at the University of York, and has worked in the UK Civil Service. He has interests in formal methods, security, reliable and distributed systems, and operating systems.

Richard Paige

Richard Paige is a lecturer in the High-Integrity Systems Group, Department of Computer Science, University of York. He is interested in object-oriented development, formal methods, security, distributed systems, and compilers. He completed his PhD at the University of Toronto in 1997.

2.1.3 Transfer symbols

The two transfer symbols, TRANSFER IN and TRANSFER OUT allow a large tree to be broken up into multiple pages; it also allows for the analysis to be broken into more manageable parts, and for duplication to be reduced.

2.2 Probabilities

Once a fault tree has been constructed, then we can consider calculating probabilities of failure. Later (in Section 4.3) we will explain that in most cases, we cannot use probabilistic analysis in computer security. The interested reader is directed to the usual Fault Tree Handbook [1], or to one of the many tools that support this automatically (next section).

Essentially, for each event, we attempt to assign a probability. For primary events, this probability must be derived from external information. For all other events, their probability is calculated from their immediate descendant events and the intervening gate.

For example, given two independent events, *A* and *B*, with probabilities *P(A)* and *P(B)* respectively, the probability of the event *P(A* AND *B)* is simply *P(A) P(B)*. Similar rules apply for determining the probability of *P(A* OR *B)* and the other gates.

If we have information concerning the probability of the primary events, then we can use such rules to determine the probability of the root event. An important issue here concerns the measurement of probability. How do we obtain that information? How accurate is it? In the case of hardware, the 'bathtub' profile is reasonably well-understood: in mass-produced components, manufacturers can often provide relatively accurate information on the failure rates. One-off systems are much harder to analyse.

It is difficult to assign probabilities to many aspects of software systems due to their more discrete, non-linear nature. We discuss the concept of 'failure' in the context of software (or discrete hardware) further in Section 4.3.

2.3 Tool support

FTA is time-consuming and sometimes difficult to manage. Tool support is necessary to enable the analyst to apply FTA to anything larger than a tiny system.

There are a number of approaches to supporting FTA. At its simplest, drawing programs such as Visio (or, as in this paper, a set of LATEX and PSTricks macros) can be used. These allow the user to apply simple picture editing to FTA.

More complex tools offer sophisticated graphical editing facilities tailored specifically to FTA, as well as evaluation features, such as calculating probabilities, im-portance of components, *etc*. A list of such tools can be found on the WWW at the University of Maryland [2].

The project at CESG referred to earlier used a bespoke program that read in a 'node-and-edge' graph description of the fault tree, and generated output that was used as input to a graph generator.

3. Related work

There has been some related work on modelling of secure systems and security properties. Little has focussed on the application of safety techniques to security; the aforementioned references [3, 4] are among the few approaches in the literature.

Work that is similar to ours is that of Schneier on *attack trees* [5]. Attack trees are a hierarchical graphical notation designed to model potential attacks and threats against a system with security constraints. Analytic techniques — similar to the probabilistic

techniques used with fault trees — can then be applied to the attack trees to determine, e.g. the likelihood of a specific class of attack succeeding. Attack trees are not intended to be used to model systems, rather, the borderline between a secure system and the outside world. They are also applied in Moore et al. [6] for information security and survivability.

Fronczak's work [7] is probably the closest to our work. It too deals with fault trees and security. However, it mostly concerns microprocessors and deals with either software or hardware faults alone. This can be justified if protecting against exactly one fault. Furthermore, the paper discusses making FTA tractable for software systems by design, i.e. incorporating design features so that branches of the fault can be considered and discarded (if appropriate) during analysis, rather than doing full tree coverage.

Winther et al.'s complementary approach applies HAZOP (HAZard and OPerability) studies to security problems [8]. A HAZOP study is (like FTA) a systematic analysis; specifically, HAZOPs looks for how deviations from the design specification of a system can arise, and whether those deviations can result in hazards. Winther et al. found the need to add specific keywords for security hazards.

An abuse case [9] is a security requirements analysis technique, particularly useful for developing assurance arguments. This is a similarity to our work on applying fault trees, which are also designed for developing arguments; this similarity is identified by McDermott [10]. However, there are substantial differences. Abuse cases are intended to be applied throughout the development process. A key difference is that FTA is based on refinement of individual undesirable events, whereas abuse cases refine interactions. It is also claimed that abuse cases are more intuitive than FTA.

A misuse case [11] is an attempt to extend the UML use case diagramming and template notation for describing functions that a system should not be able to perform. The extension also adds a notion of a mis-actor — actions that may either willingly or unintentionally misuse the system. Misuse case diagrams are mixed with use case diagrams, and additional stereotyped relationships, e.g. *prevents* and *deducts*, are added so as to represent potential security flaws and how these flaws might be prevented or annulled. Effectively, misuse cases are just use cases from a different perspective — that of a misuser. It is not yet clear whether the additional diagrammatic complexity that misuse cases introduce into graphical models is helpful. The distinctions between abuse and misuse cases appear to be minimal in intent; misuse cases provide a notational extension of UMLs use case diagrams, whereas abuse cases do not define any such extension.

UMLsec [12] is a profile of UML for modelling and developing secure systems. It extends elements of UMLs diagramming notations for precisely modelling security-relevant information. A use case-driven iterative process is provided with UMLsec that makes use of key diagramming elements like activity diagrams, class diagrams, sequence diagrams, and statecharts. Interestingly, UMLsec also adds elements of goal-directed modelling [13]. A security goal tree is developed alongside the system models; the tree and models are usually developed iteratively. The UML diagrams — particularly class and sequence diagrams — are extended with conditions obtained from goal trees and requirement modelling.

4. Security FTA

So far, we have discussed FTA in the form described in the Fault Tree Handbook [1]. In this section, we describe how FTA can be used as part of the design and analysis of security-critical systems.

We envisage that a system designer may at first use FTA to analyse the system requirements at a high level of abstraction as part of the risk

analysis. Later both the designer and a system certifier will use FTA on that system to identify issues that require further engineering to ameliorate a risk, or to justify approval of the system.

It is not intended as a one-method approach to security engineering, but as a supplement to existing techniques.

Our outline method is thus:

(1) Identify undesired (root) events that can happen.

(2) Starting at each root event, construct a fault tree.

(3) Analyse the fault tree(s).

4.1 Identify root events

A system requirements document may mandate particular properties, e.g. "only legitimate users can read patient documents". We then create the root event "an unauthorized user reads a patient document": given our system requirements, this is clearly an 'undesired' event.

Other broad classes of root events that may be of interest here are:

- An enemy reads secret information.
- An enemy masquerades as a legitimate user.
- An enemy forges a statement.

[Here, we use the term 'enemy' to mean some agent, whether a human user or computer program, who is unauthorized to carry out that particular action. More generally, this raises issues about how to identify and describe which agents are permitted to access (read, change, delete, etc.) particular resources.]

4.2 Construct fault tree

Starting at a root event, an analyst must use detailed knowledge of how the system is constructed to derive the fault tree. We stop constructing new events when all leaf events are primary (i.e. there are no uncompleted INTERMEDIATE events).

The tree should be constructed (ideally) in a breadth-first approach: this allows a balanced view of the system. This is an example of one of the rules that have been accumulated in the Fault Tree Handbook. There are a number of basic rules suggested for fault tree construction [1, Section V.7]:

Ground Rule I: Write the statements that are entered in the event boxes as faults; state precisely what the fault is and when it occurs.

Ground Rule II: If the answer to the question, "Can this fault consist of a component failure?" is "Yes," classify the events as a "state-of-component fault". If the answer is "No," classify the event as a "state-of-system fault."

This separates the faults into 'an individual component fails' for some reason (a state-of-component fault), e.g. "a valve jams open"; and "something outside the component causes its failure" (a state-of-system fault), e.g. "the room fills with water".

No Miracles Rule: If the normal functioning of a component propagates a fault sequence, then it is assumed that the component functions normally.

Complete-the-Gate Rule: All inputs to a particular gate should be completely defined before further analysis of any one of them is undertaken.

No Gate-to-Gate Rule: Gate inputs should be properly defined fault events, and gates should not be directly connected to other gates.

Many faults in this context are likely to be actions by an enemy. Some may view this enemy to be a component of the system in this context (as in the case for some process algebra approaches to security protocol analysis [14]). Fundamentally, the fault tree shows how a sequence of events leads to the root event of concern.

A question for the application of FTA to security-critical systems relates to the rules above: are they useful in that context? Can this

be demonstrated? This is difficult to ask, especially since the rules are anecdotal even for safety-critical systems analysis. At first look, the rules offer guidance for the *systematic* construction of the FT. The most difficult is 'Ground Rule II': the notion of a 'system' and a 'component' not completely clear.

4.3 Analysis

In Section 2.2, we very briefly discussed the probability calculation for a root event given information about the primary events. This is sensible when good data is available concerning the likelihood of failure of a given component.

However, in computer security, in common with many failure problems in computer software (and some types of hardware), it is difficult to assign useful probabilities to the events: the discrete, non-linear nature of these systems makes a complete nonsense of the concept.

Instead, the analyst must carry out a risk analysis [15]. The fault tree provides most of the raw data: it tells the analyst *how* the system fails, given the primary events (i.e. the BASIC, UNDEVELOPED and EXTERNAL events).

Essentially, the analysis of a fault tree for a security-critical system provides information about the interactions by which the system fails. In this sense, the FTA is also an 'attack handbook' — it identifies routes to obtain information contrary to the system security requirements. The system analyst must decide if any of those routes are plausible, or if they are unrealistic.

This last point deserves further explanation: Consider systems that use symmetric cryptography. The system may be perfectly implemented (i.e. it does not introduce any implementation faults — a rarity in real systems), but during the operational lifetime of that system, the symmetric algorithm may be effectively broken. That is, given some reasonably obtainable information, an enemy can 'break' the algorithm sufficiently quickly to obtain valuable information.

The FTA provides a communication mechanism between the system designer and the system certifier. The designer tries to show that there are no plausible attacks on the system, while the certifier attempts to show that the designer's arguments are wrong, or that the FTA is an incorrect analysis of the system.

5. Example

The example in this section is a very small system, and we have left much unresolved. A real implementation would have possible faults due to the implementation, as well as protocols in use. However, it is useful for illustrative purposes; moreover, this paper is inspired by the application of this technique to a real system.

Our example concerns software package signing: an increasingly common application of

Figure 4: Example FTA for software package signing.

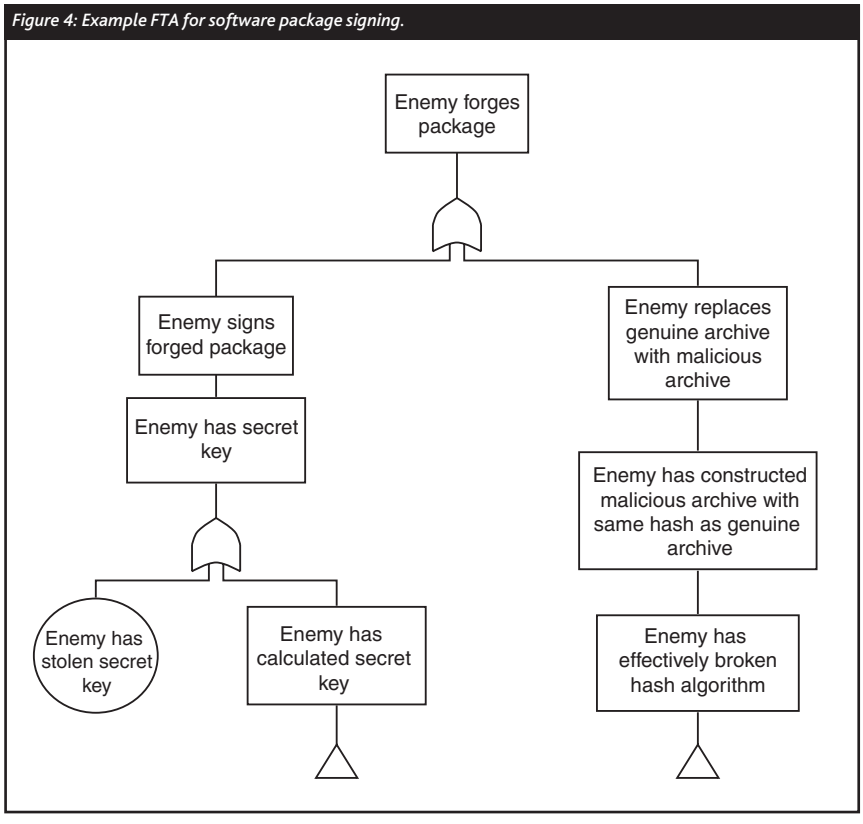


Figure 5: Example FTA for software package signing: development of 'Enemy has calculated secret key' event.

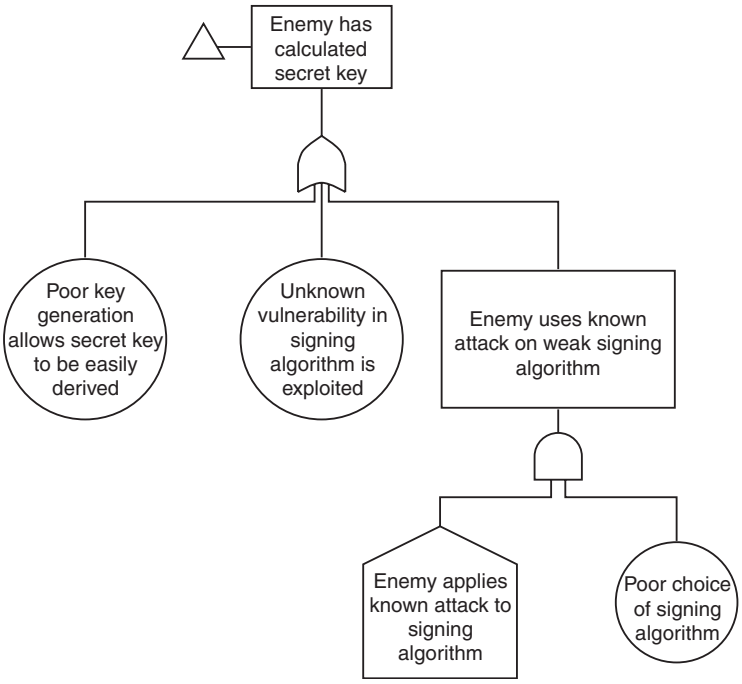
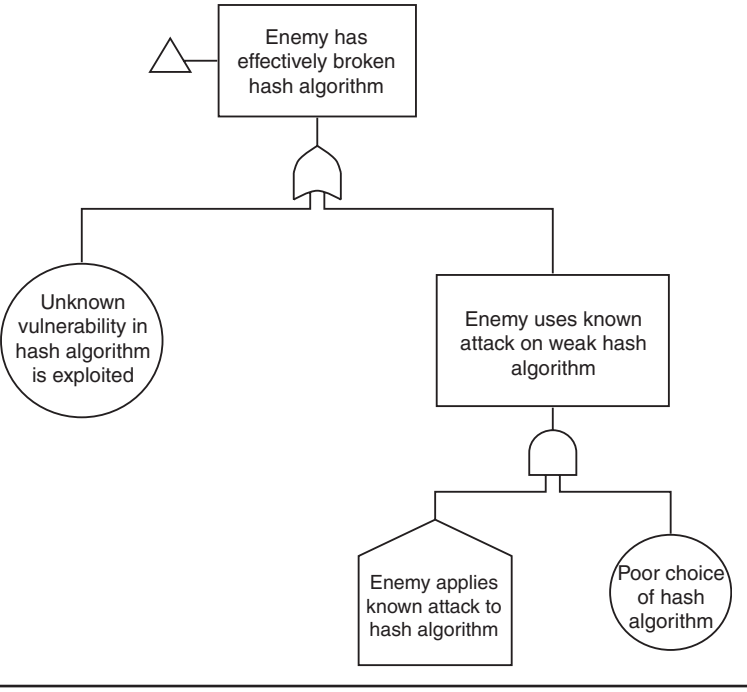


Figure 6: Example FTA for software package signing: development of 'Enemy has effectively broken hash algorithm' event.



cryptography given the distribution of software over networks.

Suppose there is a method of signing software packages, where the package consists of an archive (e.g. a `tar` archive) and a digital signature. Typically, the digital signature is calculated from a hash (of the archive) and a secret key: this signature can be checked against the (published) public key. The security of the system is in the two facts that

- it is computationally difficult to choose the input to a hash algorithm that produces a particular output, and _
- it is computationally difficult to derive the secret key from the public key.

A FTA for such a system is shown in Figure 4.

So what does Figure 4 tell us? Reasonably succinctly, we can see that any of the three leaf events in this figure would lead to the root event, i.e. that the enemy can forge packages. This itself may lead into a fault tree concerning, say, system compromise.

Looking at the three leaf events from left to right, the second and third are the two 'computationally difficult' cases. We expand on these in Figures 5 and 6.

The first event could be developed further also: this is where some attacks on such programs as SSH and PGP arise (and is the reason for requiring a passphrase to access secret data). Rather than attack the algorithms, the enemy attempts to obtain the secret key directly from the users' computer.

One might reasonably conclude from this analysis that provided strong algorithms are chosen then the major risk is from user error (e.g. not properly protecting secret keys).

6. Discussion

The approach presented in this paper requires considerable effort to apply. The project at

CESG where this approach was devised resulted quickly in a fault tree which was best displayed on a poster larger than A0, unlike our substantially smaller examples. However, a real system has much more necessary complexity, and a useful analysis may need a finer detail of resolution.

Despite the size of the fault tree, its structure provided valuable insights that led to design improvements. In particular, identifying sets of a small number of primary events suggested fixes to threats in the system, while identifying other issues as unlikely to be easily exploited.

The major benefit of this approach (as far as security is concerned) is that it explicitly identifies the relationship of events to each other, including the root fault events.

As described in Section 4.3, fault trees offer a realistic framework around which to make a security argument: the designer who claims that his system is sufficiently secure can use the fault tree to identify how antecedent faults are unlikely to cause a root failure. The certifier can use the fault tree by identifying invalid arguments or invalid analysis (e.g. omitted events).

The problem here is that sufficiently competent staff are required to determine if attacks are plausible. Thus we emphasise that FTA is a useful *aid* to the analysis of security-critical systems, and not a substitute for proper and knowledgeable design.

For example, the analysis of cryptographic algorithms and protocols is directly related to such FTA, but themselves require substantial knowledge and skill.

When plausible attacks are detected, the system should be redesigned or adapted to fix this vulnerability, after which the FTA can be repeated. Ideally, the FTA would indicate an appropriate fix (e.g. redundancy of hardware).

Future work should include case studies based on real systems with some security requirements: it is hoped that this will identify particular structures of interest, similarly to software design patterns.

In Section 4.2, we touched on justifying the 'rules' for construction of fault trees. The future work above could also attempt to determine the validity of those rules, as well as identifying other possible 'good practice'.

7. Conclusion

This paper has briefly presented fault trees in their classical application of safety analysis. We claim that FTA is a technique to be applied to both the design and analysis of systems with security requirements.

In some sense, this work broadens the work by Helmer et al. [4]. There is no reason not to apply FTA to broader security systems than only intrusion detection. By illustrating some small examples within the security domain, this paper demonstrates the applicability of FTA to security-critical systems.

Acknowledgements

The ideas in this work came about while the first author was employed by CESG, UK. This paper has been written at the University of Plymouth. Thanks are due to Steven Furnell (Network Research Group, University of Plymouth) and the members of the University of York Secure Network group for their comments on this paper.

References

[1] US Nuclear Regulatory Commission, Fault Tree Handbook, NUREG-0492 (January 1981).

[2] Fault tree analysis programs, http://www.enre.umd.edu/ftap.htm, materials and Engineering Department, University of Maryland.

[3] Anderson, R.J., 2000. *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley, 2001.

[4] Helmer, G., Wong, J., Slagell, M., Honavar, V., Miller, L. and Lutz, R., 2001. A software fault tree approach to requirements analysis of an intrusion detection system. *1st Symposium on Requirements Engineering for Information Security 2001*, Indianapolis, Indiana, USA, 2001.

[5] Schneier, B., 1999. Attack trees: Modeling security threats, *Dr Dobb's Journal*, 1999.

[6] Moore, A., Ellison, R. and Linger, R., 2001. *Attack modeling for information security and survivability*. Tech. Rep., Carnegie Mellon Software Engineering Institute (2001).

[7] Fronczak, E.L., 1998. A top-down approach to high-consequence fault analysis for sofrware systems. *Proceedings of the 16th International System Safety Conference*, 1998.

[8] Winther, R., Johnsen, O.-A. and Gran, B.A., 2001. Security assessments of safety critical systems using HAZOPs. *Proceedings of SAFECOMP 2001*, 2001.

[9] McDermott, J. and Fox, C., 1999. Using abuse case models for security requirements analysis. *Proc. COMPSAC 1999*, 1999.

[10] McDermott, J., 2001. Abuse case-based assurance arguments. *Proc. Computer Security Applications Conference 2001*, 2001.

[11] Sindre, G. and Opdahl, A., 2000. Eliciting security requirements by miuse cases. *Proc. TOOLS Pacific 2000,* IEEE Press, 2000.

[12] Jurjens, J., 2002. Using UMLsec and goal trees for secure systems development. *Proc. Symposium on Applied Computing (SAC) 2002*, ACM Press, 2002.

[13] Chung, L., 1993. Dealing with security requirements during the development of information systems. *Proc. CAiSE'93, no. 685 in LNCS,* Springer-Verlag, 1993.

[14] Lowe, G. and Roscoe, B., 1997. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, Vol. 23 (10), 1997, pp. 659-669.

[15] Sommerville, I., 2001. *Software Engineering*, 6th Edition, Addison Wesley, 2001.

Correction

There is an error in Sheng Zhong's paper, 'A Practial key management scheme for access control in a user hierarchy, *Computers & Security*, vol. 21, no. 8.

Section 3.2.2, bullet (3) reads
'For each C_i, T computes $P_i \leftarrow P_i \oplus L_i$.';
it should read
'For each C_i, T computes $P_i \leftarrow K_i \oplus L_i$.'.

Computers & Security apologizes to Sheng Zhong for this error.