

A Top-Down Approach to High-Consequence Fault Analysis for Software Systems

Edward L. Fronczak; Sandia National Laboratories; Albuquerque, New Mexico

Keywords: software-system safety, software-system security, fault tree analysis, failure modes and effects analysis, fault tolerance

Abstract

Fault Tree Analysis can be used to identify hardware failures that will result in high-consequence events in software systems. It can also be used to develop a code inspection procedure to find software design errors that can cause these high-consequence events in the absence of hardware failures. Features need to be added to the software system to make the Fault Tree Analysis tractable.

Fault Trees can be used during the software system design to establish that an appropriate combination of security features is correctly implemented. This is much better than merely implementing a checklist of good design practices.

A top-down approach based on Fault Tree Analysis is more effective and more efficient than bottom-up approaches based on Failure Modes and Effects Analysis for identifying high-consequence faults in microprocessor-based systems.

This paper contains a detailed example demonstrating the top-down analysis approach. A brief comparison of top-down and bottom-up approaches is also included.

Introduction

Software systems are used in safety and security applications where undesired high-consequence states may occur. The software code may be inadvertently designed to allow these states to be reached. Even if software code is fault-free, hardware failures can alter a value in memory, possibly where the code itself is stored, causing the computer system to reach an unacceptable state. A common design goal in these kinds of systems is to eliminate high-consequence software coding failures and to design so that single hardware failures won't cause a high-consequence event.

Since the large number of states software systems can reach makes exhaustive testing impossible, analytical methods are needed to determine whether or not these design goals have been achieved. Either a bottom-up approach, based on Failure Modes and Effects Analysis,

or a top-down approach, based on Fault Tree Analysis may be used. This paper compares the two approaches and concludes that top-down analysis is more effective.

Tolerance to single hardware failures can be achieved by building duplicate systems and comparing their outputs. Where this approach is not possible because of cost, power consumption, or some other design constraint, a detailed system analysis that goes down to the bit level may be required. The example in this paper shows how to do the detailed analysis.

This paper describes and demonstrates a top-down methodology, based on Fault Tree Analysis that has been used to identify potential high-consequence faults in microprocessor-based systems. The key to making the Fault Tree Analysis tractable is to effectively incorporate appropriate design features such as software path control and checksums so that complicated branches of the fault tree can be terminated early. The analysis uses simplified flow diagrams depicting relevant code elements. Pertinent sections of machine language are then examined to identify suspect hardware.

This approach has been used to analyze systems with high-consequence events such as 1) allow unauthorized access, 2) inadvertently divulge classified information, 3) inadvertently detonate a mine, and 4) report a system is in a safer than actual state. In every case this methodology has revealed potential failures, undetected by previous analysis, that would cause these events.

Fault tree analysis is sometimes used to examine software code written in a high-level language. Unless there is a guarantee that safety and security features are compiled into the machine code correctly, analysis must be performed using the machine language.

Comparison of Fault Analysis Methodologies

If detailed analysis of the software system is necessary, the analyst must decide how to proceed. The basic approach to bottom-up and top-down analysis is outlined below. A comparison of these methodologies shows why one would expect a top-down approach to be more effective and efficient.

Bottom-up High-Consequence Fault Analysis

Methodology: A widely used methodology for High-Consequence Fault Analysis is based upon Failure Modes and Effects Analysis (FMEA). This bottom-up analysis is performed using the following procedure:

- 1) Identify high-consequence events.
- 2) Determine hardware failure modes that will be considered.
- 3) Determine critical parts of the system.
- 4) Assume a particular hardware failure.
- 5) Determine whether or not this hardware failure will result in one of the identified high-consequence events.
- 6) Repeat steps 4 and 5 until all hardware failures of all critical parts have been considered.

Once critical parts of the system (failure might result in a high-consequence event) are determined, the analysis cycles through each failure mode of each critical piece of hardware. To be done thoroughly, each failure mode must be considered for every possible state of the system. In practice this is never done since the number of combinations of failure modes and system states is prohibitively large. This means that there is always some question about both the quantity and quality of analysis performed. Results are typically reported in the form of a table such as in Figure 1 where the ability of each component failure to cause each top-level fault (F1, F2, and F3 in this example) is noted by some brief entry. If the effect of the failure is a high-consequence event, some supporting text is typical. However, if the analyst sees no resulting top-level fault, an entry such as "No effect" with little or no supporting text is typical.

Top-down High-Consequence Fault Analysis

Methodology: The top-down High-Consequence Fault Analysis methodology described here is based on Fault Tree Analysis (FTA). It is performed using the following procedure:

- I) Identify high-consequence events.
- II) For each high-consequence event:
 1. Assume the software is fault-free
 - a) Assume the high-consequence event (top-level fault) occurs
 - b) Construct a hardware fault tree integrating hardware and software interactions
 - 1) Use simplified system block diagrams depicting relevant components
 - 2) Use simplified software flow diagrams depicting relevant code elements
 - 3) Incorporate software features to facilitate the analysis
 2. Assume the hardware is fault-free
 - a) Assume the high-consequence event (top-level fault) occurs
 - b) Construct a software fault tree using simplified flow diagrams
 - c) Use the software fault tree to develop a code inspection procedure to complete the analysis.

The analysis proceeds by building the hardware fault trees, creating simplified block diagrams and flow charts as needed. The software flow charts serve as a guide to locating critical areas in the program memory that must be examined. Hardware involved in these critical code operations make up the list of "usual suspects." To construct a fault tree that identifies single faults, first assume that the high-consequence event occurs. Next determine the immediate and necessary conditions that must occur for this top-level fault to occur. These conditions are themselves lower-level faults. Repeat this process for each lower-level fault until each branch of the fault tree terminates. Fault tree branches terminate for one of the following reasons:

- 1) A point is reached where you have an "AND" gate with two independent hardware failures as inputs. Sometimes an argument is made that two independent hardware failures exist even though one of them may not be explicitly identified. This may happen, for example, when a subroutine is used to check for bit failures. For the bit failure to be undetected the bit has to fail and some unidentified bit in the subroutine has to fail.
- 2) A point is reached where an argument is made that a fault is so unlikely that it is not worth pursuing.
- 3) A single hardware failure leading to a security fault is identified.

Establishing independence of failures is often difficult. We really are looking for failures that are not strongly correlated. Here we will identify two different hardware failures and remember that we still need to determine how correlated the failures are.

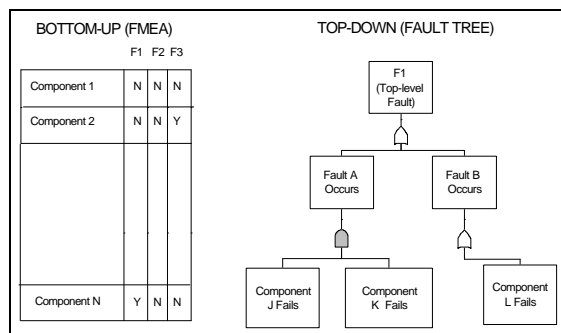


Figure 1. Comparison of Bottom-up and Top-down Fault Analysis Outputs

This approach can be modified slightly to identify multiple faults by not terminating fault tree branches until several faults are identified that must occur to cause the next higher fault. Time is better spent, however, looking for single faults until one is convinced they have all been identified since the likelihood of single faults occurring tends to be much greater than the likelihood of multiple faults occurring.

The fault trees are typically constructed as tree diagrams with AND gates and OR gates connecting the fault entries such as in Figure 2. Here High-Consequence Fault 1 (F1) occurs if either Fault A or Fault B occurs. Fault A occurs only if both Component J and Component K fail. Fault B occurs if Component L fails. The failure of Component L is a fault since its occurrence results in F1.

The software fault tree is constructed assuming that the hardware is fault-free. All identified software design faults will be corrected.

The Key: Design Features that Facilitate Analysis: As the fault tree develops, one often sees that a small software modification can eliminate large sections of hardware from concern. An important example of this is the use of some form of program flow control to guard against inadvertent jumps during operation. Rather than examine every bit failure to see if it will cause an inadvertent jump into a critical routine, a time consuming if not impossible task for large systems, assume the inadvertent jump occurred and do something to detect it and stop the critical routine from running. By noting the normal path through the software to reach the critical routine and adding a few lines of code at strategic points along the way to modify a flow control word, we can detect an inadvertent jump and fail safe.

Other examples of design modifications to aid analysis suggest themselves during the analysis. Software modifications are easy to make and simplifying the analysis serves the designer's, the analyst's, and the customer's best interests. The ability of the customer to understand the high-consequence fault tree analysis is an added and not inconsequential benefit.

Comparison of the Two Methodologies:

Bottom-up: Some hardware is determined to be non-critical before a detailed analysis is performed. If a mistake is made here, critical hardware will not be analyzed.

Top-down: Hardware is only deemed non-critical by looking at the full analysis. If it doesn't appear on the fault tree, it isn't critical.

Bottom-up: In order to be thorough, every failure mode of every component of critical hardware must be considered during every system state. This is not practical for large systems.

Top-down: Only failure modes and system states that appear in the developed fault tree are pertinent. It is not necessary to discuss every failure mode of every component during every system state. Large systems can be analyzed.

Top-down: Can be used in an analysis that considers 2 or more failures that will cause a security compromise.

Bottom-up: Analysis effort considering even 2 failures that will cause a security compromise is intractable.

Bottom-up: Details of the analysis are omitted in the write-up, especially when no link between a failure mode and the high-consequence event is found.

Top-down: The fault tree is the detailed record of the analysis.

Bottom-up: Presentation is lengthy because all failure modes must be accounted for.

Top-down: Presentation is concise and easily understood. Only pertinent failure modes are included.

Bottom-up: The relationship between each fault and the top-level security compromise is typically described with text and is hard to follow.

Top-down: The relationship between each fault and the top-level security compromise is clear by looking at the fault tree.

Bottom-up: Analysis cannot proceed until the design is complete.

Top-down: Fault trees can be incorporated as part of the design methodology.

Bottom-up: Not sure how to account for software and hardware interactions.

Top-down: Incorporating hardware-software interactions is basic to the methodology.

Bottom-up: Not useful for software code fault analysis.

Top-down: Fault analysis of software code is possible.

There appears to be no downside to using this top-down approach, unless you're getting paid by the pound for doing the analysis. An analyst will get better results

and get them more quickly using the top-down approach.

Example: Allow Unauthorized Access

An analysis of the following simple security system shows how a fault analysis is performed. First, we look at a design that has no fault tolerance features and identify, through the analysis, hardware failures that will compromise system security and types of hardware failures that may exist but which might be very difficult to analyze. Next we will show how software features can be incorporated to address the identified concerns. Finally we will use a software fault tree analysis to derive a code inspection procedure that can be used to show whether or not the code design is free of security faults.

System Description: A security guard observes activity at a gate. When someone approaches the gate, she must either determine that they are good guys and let them in, or that they are bad guys and issue an alarm. She controls activity by selecting among three keypad commands: Open, which opens the gate; On, which turns on an alarm; Off, which turns off the alarm. The gate automatically closes after someone passes through.

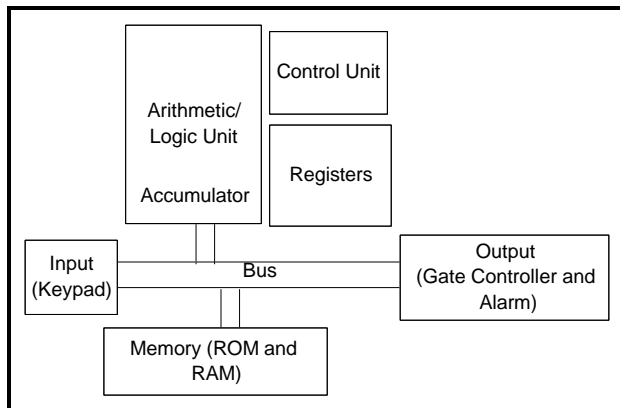


Figure 2. System Block Diagram

Figure 2 shows a block diagram of the microprocessor system used to control operations at the gate. The software program is stored in Read Only Memory (ROM) and uses the Random Access Memory (RAM) as a work space. The microprocessor consists of an Arithmetic Logic Unit (ALU), a Control Unit, and some programmable registers. One of the registers, the Program Counter, contains the address of the next program instruction to be performed. Another, the Flag

Register, is used in branching instructions. Data is transferred from the Input Device (keypad) to the microprocessor and from the microprocessor to the Output Device (gate controller) over the system bus through another register, the Accumulator, which is part of the ALU.

For this analysis, we assume that the keypad and gate controller are free of security faults.

Identify High-Consequence Events: While no system malfunction is desired, some malfunctions, such as inadvertently opening the gate, compromise security. When bad guys show up at the gate and the guard selects the alarm command, the last thing she wants to see is an opening gate. This system failure is so costly that we do not want the presence of a single bit error to result in the gate being inadvertently opened. Other malfunctions, such as inadvertently sounding the alarm or failing to open the gate given an Open command, may be inconvenient but do not compromise security. The high-consequence fault analysis focuses on those system failures that compromise security.

For this system, the high-consequence event of concern is "Send Open message inadvertently."

Hardware Fault Tree (Assuming Fault-Free Software): Develop the fault tree by assuming that the undesired event has occurred, and work backward from that event looking for hardware failures that might be to blame.

It will be useful to have a software flow chart, such as in Figure 3, that shows how the Open message is sent to the gate controller when the guard selects the Open command at the keypad. When the guard selects a command, the keypad device makes an 8-bit word available to the system bus. The two least significant bits are used to identify the selected command as shown below:

- 01 is (alarm) Off
- 10 is (alarm) On
- 11 is Open (the gate).

The other six bits may be used to carry some other information such as keypad status. A machine language instruction, IN, puts this 8-bit word onto the system bus and copies it into the accumulator. This word is then saved at RAM location MESSAGE. Some functions such as checking the keypad status are then performed. Then the content of MESSAGE is copied into the accumulator so that the selected command can be determined. Next the six bits in the accumulator other than the two command bits are set to zero. The content of the accumulator is now compared with the three two-bit command values and the program branches to the appropriate routine. If the Open branch is taken, the

output message is copied from memory into the accumulator. A machine language instruction, OUT, then puts the contents of the accumulator onto the system bus where it is available to the gate controller.

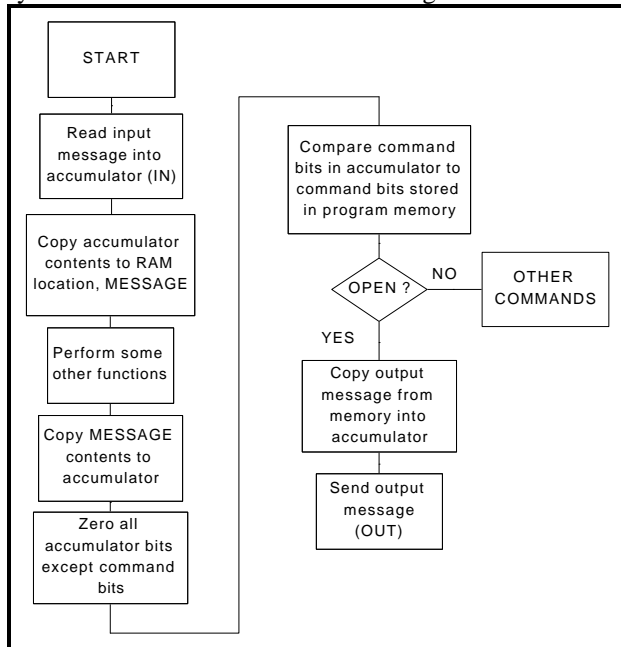


Figure 3. Software Flow Diagram

Referring to the example fault tree in Figure 4, the top event,

- 1) send open message inadvertently is equivalent to
- 2) OUT instruction is performed with an Open message in the accumulator when the Open command was not selected.

Referring to the flowchart, fault 2 can occur in only two ways:

- 3) the Open message was sent when not executing code in the Open branch, or
- 4) the Open message was sent when executing code in the Open branch.

- Fault 3 can happen only if
- 5) the accumulator contains the Open message, and
 - 6) an OUT instruction is executed.

We know that the Open message is copied from memory into the accumulator only in the Open branch of the software and that the only OUT instruction is in the Open branch. We may argue (incorrectly) that 5 and 6 represent two faults - one that places the Open message into the accumulator and one that causes an OUT instruction to be performed. But the Open message is just a particular 8-bit word, and we don't know whether or not that 8-bit value appears in the accumulator during

some other operation such as the result of a calculation. Since we cannot assume that the Open message appearance in the accumulator outside the Open branch is a fault, we assume that if an OUT instruction is performed, the Open message is sent.

- Fault 6 can occur if there is a
- 7) bit failure in program memory when the intended instruction is one bit different from OUT, or
 - 8) a bit failure in program memory changes an instruction forcing the system into an incorrect state.

To expand 7 any farther would require identifying every place in the software where an instruction one bit different than an OUT instruction is used. Expanding 8 is even more problematic; it requires examining the effect of every stuck bit in program memory. Instead of attempting this analysis, we assume that 7 and 8 can each occur due to single bit failures. Later, we will add software features to get rid of these faults. The analysis is much easier once the software features are added.

- Fault 4 can occur if
- 9) the Open branch is taken as the result of the designed compare, or there is an
 - 10) inadvertent jump into the Open branch.

To expand 10 would be very difficult so we assume that it can occur due to a single bit failure.

To see how Fault 9 can occur we need to look at the section of code where the compare is implemented. In this design, the compare is implemented with the series of Compare and Jump instructions below:

```

Cmp 03
JZ  Open
Cmp 02
JZ  Alarm On
Cmp 01
JZ  Alarm Off
  
```

Cmp compares the hexadecimal (base 16) operand (03, 02, or 01) to the value in the accumulator and sets the zero flag in the flag register to 1 if they are equal. JZ jumps to the location indicated in the operand (Open, Alarm On, or Alarm Off) if the zero flag is set to 1. If the zero flag is set to 0, no jump occurs and program execution passes to the next instruction.

- Fault 9 occurs if
- 11) the zero flag bit is failed to 1 or
 - 12) the Open branch is taken when the flag is OK.
- Note that Fault 11 would cause the Open branch to be taken no matter which command was selected.

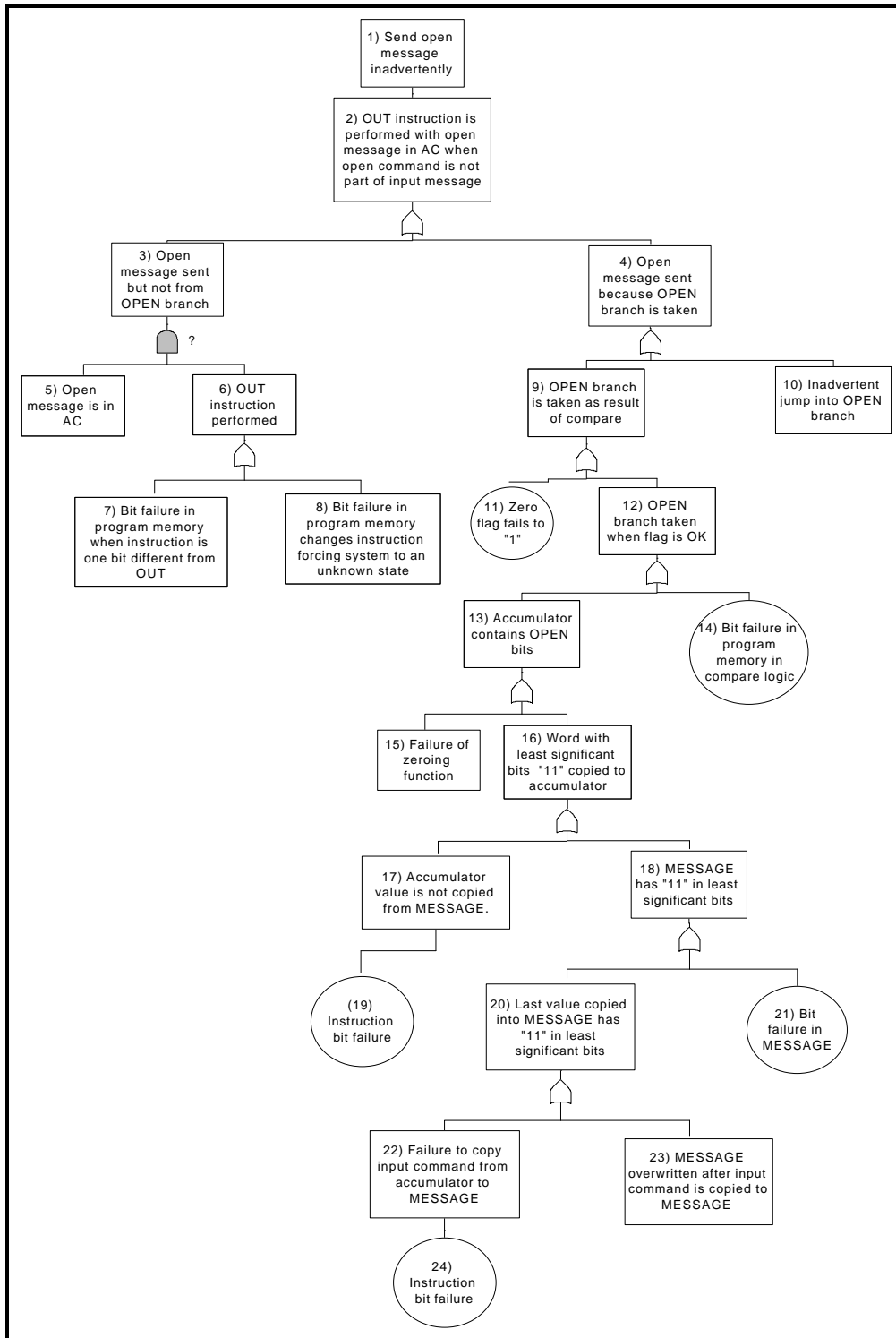


Figure 4. Hardware Fault Tree (Assuming Fault-Free Software)

Fault 12 occurs if the
 13) accumulator contains the Open command bits (03
 hex) or there is a

14) bit failure in program memory where the compare is
 implemented.

Fault 14 may occur several ways. For example, if Alarm
 On is selected, a single bit error in the first compare

operand changes Cmp 03 to Cmp 02 and instead of sounding the alarm, the gate opens. We might discover that a single bit failure in the first JZ instruction opcode bit pattern changes the instruction to JNZ, jump if not zero, causing the gate to open if either an Alarm On or an Alarm Off instruction is selected. Several other bad things can happen here. Rather than identify them all, just assume that a problem exists and plan to incorporate software features to simplify the analysis.

Fault 13 can occur if

- 15) the correct bits are copied into the accumulator, but the zeroing function fails, or
- 16) an 8-bit word with bit pattern 11 in least significant bits is copied into the accumulator before the zeroing occurs.

To see how Fault 15 might occur we look at the assembly code and note that the zeroing is implemented with the instruction "AND 03". This instruction performs a logical AND of the accumulator contents with the hexadecimal value 03 and puts the results in the accumulator. Because of a fortunate assignment of bit pattern to command, a bit failure in the operand, 03, may cause the alarm to sound when gate opening is desired, but will not cause the gate to inadvertently open. If there was a single bit difference in the opcode bit patterns for AND and OR, we would have a problem. The instruction "OR 03" would cause the gate to open when any command was selected.

Fault 16 might occur if

- 17) the accumulator value was not copied from MESSAGE, or
- 18) MESSAGE has a value with the bit pattern 11 in the command bits.

Fault 17 occurs if

- 19) there is an instruction bit failure which changes either the copy opcode or the address in the operand.

Fault 18 occurs due to either

- 20) the last value copied into MESSAGE has bit pattern 11 in least significant bits, or
- 21) bit failure in MESSAGE.

Fault 20 occurs if

- 22) there is a failure to copy the accumulator contents to MESSAGE, or
- 23) MESSAGE was inadvertently written to after the input command was copied to MESSAGE.

Fault 22 occurs if

- 24) there is an instruction bit failure which changes either the copy opcode or the address in the operand.

Rather than conduct an exhaustive, and probably exhausting, search of all possible bit failures that might overwrite MESSAGE, we assume that Fault 23 can occur due to any of several single bit failures and plan to modify the software to eliminate them.

At this point we pause in the fault tree development.

We have identified some locations (faults 11, 14, 15, 19, 21, and 24) where single hardware faults can occur that will cause the top event. The designer can build in some appropriate redundancy or make use of checksums to get rid of these. We have also stopped development of some branches (at faults 7, 8, 10, and 23) that may be difficult to analyze and for which we would like to incorporate software features that will make the analysis more tractable.

Apply Fault Tolerance Features: Incorporating features in the software is the key to making the fault tree analysis tractable. When the number of possible bit failures to consider for one fault tree branch is large, we would like a strategy where we don't have to identify all of them. Instead, we assume they exist and provide protection against them. This not only keeps the tree size small, but also helps to present an understandable case for how well you have addressed the single bit failure issue. It is desirable that a relatively uncomplicated analysis is performed so that the customer understands the analysis. Keeping the fault tree small has the advantage of making the analysis more understandable as well as allowing the analyst to focus on fewer problems enabling them to be addressed more carefully.

Examples of how fault tolerance techniques might be used to provide protection against the above identified and assumed single faults follow.

The flag in Fault 11 could be checked by adding a few lines of code to make sure that it can be set and cleared before performing the compare.

Faults 14, 15, 19, and 24 occur in program memory. A checksum of these critical sections of code might be used to detect single bit failures here.

Faults 7 and 8 also occur in the program memory. A checksum of the entire memory is needed to detect either of these single bit failures. If performing the checksum incurs an unacceptable cost (say it takes too long) it may be possible to change the format of the open

message from one word to two words which would protect against the single failure in fault 7 and make fault 8 much more unlikely to occur. Once in the unknown state, two OUT instructions would need to be performed while the correct message portions are in the accumulator.

Fault 10, the inadvertent jump into the Open branch, can be addressed by adding a protective feature that detects security compromising program flow. Considering the flow diagram for an Open command, and modifying a flow control word at several points in this flow can do this. An inadvertent jump into the Open branch would bypass one or more of these modifications and would be detected by checking the word immediately before performing the OUT instruction.

In our example, we could:

- 1) Initialize a Flow Control Word before reading the input message into the accumulator.
- 2) Modify the command compare and branch instructions as follows:

```
Cmp 00
JZ Error
Cmp 01
JZ Alarm Off
Cmp 02
JZ Alarm On
(Modify flow control word here)
Cmp 03
JZ Open
```

- 3) Check the Flow Control Word in the Open branch for the modified value then reset the Flow Control Word to its initialized value immediately. We don't want the modified Flow Control Word stored at the beginning of the next command.

Fault 21 is due to a bit failure in the RAM location MESSAGE. Multiple storage of the contents could be used here.

Fault 23 is due to an assumed but unidentified fault in the program memory. Multiple storage of the contents of MESSAGE could also be used here.

Once these strategies are implemented, the fault tree is revised so that each branch of the fault tree that ended with a single fault, now terminates with an AND gate whose inputs represent two different failures. Figure 5 shows how these modifications look.

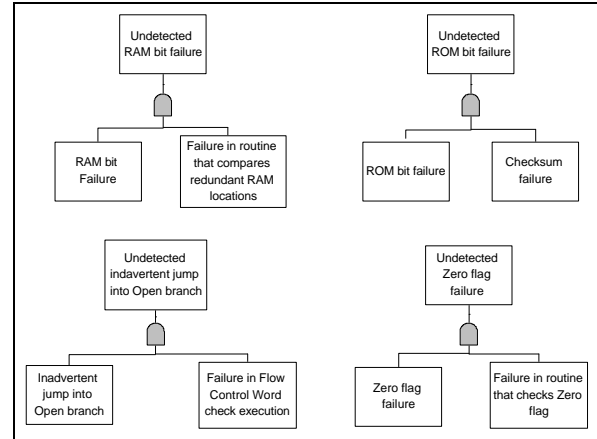


Figure 5. Fault Tree Branch Terminations

The revised fault tree documents your efforts to identify and protect against single hardware failures causing the open message to be inadvertently sent.

Software Fault Tree (Assuming Fault-Free Hardware):

For each hardware fault tree, we can construct a corresponding software fault tree assuming fault-free hardware.

Some of the higher level faults on the software fault tree are identical to those on the hardware fault tree. Lower level faults will be different since they are associated with software design mistakes instead of hardware failures.

Like we did for the hardware fault tree, use the simplified flow diagram as a guide.

For the example software fault tree in Figure 6, the top event,

- 1) send open message inadvertently,
- is equivalent to saying that an
- 2) OUT instruction is performed with an open message in the accumulator when the input message does not indicate an Open command.

Referring to the flow chart, we reason that fault 2 can occur in only two ways:

- 3) the open message was sent when not executing code in the Open branch, or
- 4) the open message was sent when executing code in the Open branch.

Fault 3 can happen only if

- 5) an OUT instruction is performed when the open message is in the accumulator.

We will assume that the open message is in the accumulator if an OUT instruction occurs outside the Open branch.

Fault 4 occurs if

- 6) the Open branch is taken as the result of the compare,
- or
- 7) there is an unintended path into the Open branch.

Fault 6 occurs if

- 8) the compare logic is good but inputs to the logic are bad, or
- 9) the compare logic is bad.

Fault 8 is equivalent to

- 10) the accumulator has the wrong value when the compare logic is performed.

Fault 10 occurs if

- 11) the value to compare is copied to the accumulator from the wrong memory location (or not copied at all) or,
- 12) MESSAGE has the wrong value when it is copied to the accumulator.

Fault 12 occurs if

- 13) the logic that performs the write to MESSAGE is bad, or
- 14) there is a write to MESSAGE after the input message is correctly written to MESSAGE.

We can use this fault tree to derive a code inspection procedure. If the code has certain easily verifiable properties that are derived from the fault tree it can be deemed "safe." That is, in the absence of hardware failures, the high-consequence event will not occur. The code inspection is the software fault tree analog of the software features that we added for the hardware fault tree.

For the example, we must verify that:

- 1) There are no OUT instructions outside of the Open branch (fault 5);
- 2) There are no unintended paths into the Open branch (fault 7);
- 3) The compare logic is not bad (fault 9);
- 4) The contents of MESSAGE are copied into the accumulator just before the compare logic is executed (fault 11);
- 5) The input command is copied to MESSAGE (fault 13); and,
- 6) There are no writes to MESSAGE after the input command is written to MESSAGE (fault 14).

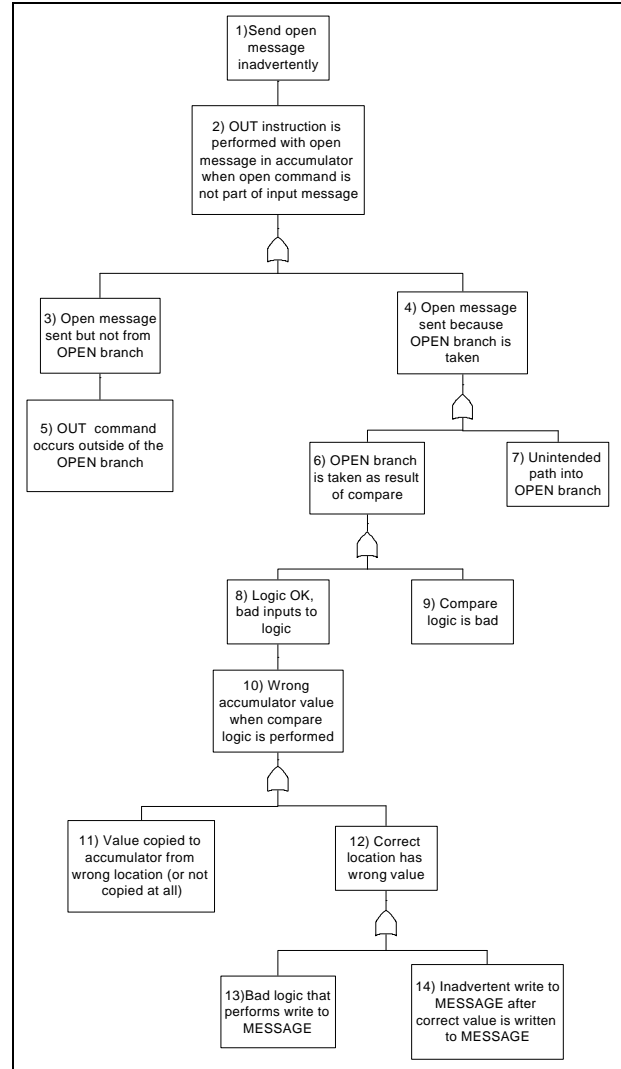


Figure 6. Software Fault Tree (Assuming Fault-Free Hardware)

Reference

- 1) N. H. Roberts, W. E. Vesely, D. F. Haasl, F.F. Goldberg. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Washington, D. C. January 1981.

Biography

Edward L. Fronczak, Sandia National Laboratories, PO Box 5800, Albuquerque, NM 87185, USA, telephone – (505) 844-5215, facsimile – (505) 844- 9971, e-mail – elfronc@sandia.gov.

Mr. Fronczak performs high-consequence fault analysis on microprocessor-based safety and security systems at Sandia National Laboratories. He has a Masters Degree in Electrical Engineering from the University of New Mexico.