FET 2003 - 5th IFAC Conference on
Fieldbus Systems and Their Applications
Aveiro, Portugal, July 7-8, 2003

CRITICAL MESSAGE INTEGRITY OVER A SHARED NETWORK

Jennifer Morris and Philip Koopman

ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
jenmorris@cmu.edu, koopman@cmu.edu

Abstract: Cost and efficiency concerns can force distributed embedded systems to use a single network for both critical and non-critical messages. Such designs must protect against masquerading faults caused by defects in and failures of non-critical network processes. Cyclic Redundancy Codes (CRCs) offer protection against random bit errors caused by environmental interference and some hardware faults, but typically do not defend against most design defects. A way to protect against such arbitrary, non-malicious faults is to make critical messages cryptographically secure. An alternative to expensive, full-strength cryptographic security is the use of lightweight digital signatures based on CRCs for critical processes. Both symmetric and asymmetric key digital signatures based on CRCs form parts of the cost/performance tradeoff space to improve critical message integrity.

1 INTRODUCTION

Distributed embedded systems often contain a mixture of critical and non-critical software processes that need to communicate with each other. Critical software is "software whose failure could have an impact on safety, or could cause large financial or social loss" (IEEE, 1990). Because of the high cost of failure, techniques such as those for software quality assurance described in IEEE Std 730-1998 (1998) are used to assure that such software is sufficiently defect free that it can be relied upon to be safe. However, such a process is expensive, and generally is not applied to non-critical system components.

In a typical safety-critical transportation system, such as in the train or automotive industries, it is generally assumed that critical components will work correctly, but that non-critical components are likely to have defects. Defects in non-critical components could, if not isolated, compromise the ability of critical components to function. Thus, the simplest way to assure system safety is to isolate critical and non-critical components to prevent defects in non-critical components from undermining system safety. Such separation typically involves using separate processors, separate memory, and separate networks.

Separation of critical and non-critical networked messages (i.e., through the use of separate buses) can double the required network costs, both in cabling and in network interface hardware. There is strong financial incentive to share a single network between critical and non-critical message traffic. But, when such sharing occurs, it is crucial that there be assurance that non-critical network traffic cannot disrupt critical network traffic. This paper assumes that non-critical network hardware is designed to be fail-safe. For example, in railroad signaling lack of message delivery leads to a safety shutdown, so safe operation is viable with off-the-shelf networking hardware. But other challenges remain to implementing such systems.

A significant challenge in mixed critical and non-critical networks is ensuring that a non-critical process is unable to *masquerade* as a critical message sender by sending a fraudulent critical message. Masquerading is considered malicious if an internal or external attacker intentionally represents itself as a different entity within the system; however, masquerading may also occur due to non-malicious transient faults and design errors that inadvertently

cause one node or process to send a message that is incorrectly attributed to another node or process.

A *software defect masquerade fault* occurs when a software defect causes one node or process to masquerade as another (Morris and Koopman, 2003). One example is a software defect that causes one process to send a message with the header identification field of a different process. Another example is a software defect that causes one node to send a message in another node's time slot on a TDMA network. A software defect masquerade fault is not caused by transient anomalies such as random bit flips, but rather is the result of design defects (e.g., the software sends the message with the incorrect header x instead of the correct header y). Fault tolerance methods designed to catch random bit flips may not sufficiently detect software defect masquerade faults.

This paper describes six successively more expensive levels of protection that can be used to guard against masquerade faults. Rather than limiting the analysis to malicious faults, the gradations presented recognize that many embedded systems have reasonable physical security. Therefore it is useful to have design options available that present tradeoff points between the strength of assurance against masquerading faults and the cost of providing that assurance.

2 FAULT MODEL

Network fault detection techniques vary from system to system. Some rely solely on a network-provided message Cyclic Redundancy Code (CRC) or other message digest such as a checksum for error detection. Some critical applications add an additional application-generated CRC to enhance error detection. These techniques can be effective at detecting random bit errors within messages, but they might not detect erroneous messages caused by software defects that result in masquerading. This is especially true in broadcast-oriented fieldbuses in which applications have control over message ID fields and can send incorrect IDs due to component defects. In the worst case, these erroneous messages could lead to masquerading of critical messages by non-critical processes or by failed critical hardware nodes.

In order to determine the safeguards necessary to ensure correct behavior of critical components over a shared network, we must first understand the types of failures that can occur, as well as their causes. The strongest fault model, which includes malicious and intentional faults, assumes that an intruder is intentionally falsifying message traffic and has significant analytic abilities available to apply to the attack. Such malicious faults can only be detected by application of rigorous cryptographic techniques. Because a malicious attack is the most severe class of fault, such measures would also provide a high degree of fault tolerance for software defect masquerade faults. But full-strength cryptographic techniques are cost-prohibitive in many embedded systems. In systems for which malicious attacks are not a primary concern, lighter-weight techniques that protect against accidental (non-malicious) faults due to environmental interference, hardware defects, and software defects are highly desirable.

The simplest accidental failures come from random bit errors during transmission. In general, these errors are easy to detect using standard error detecting codes such as CRCs that already exist on most networks.

Errors due to software defects or hardware faults in transmitters are more difficult to guard against. They can result in undetectable failures in message content unless application-level error detection techniques are used because faults occur before the message is presented to the network for computation of a CRC. A particularly dangerous type of error that could occur is an incorrect message identifier or application-level message source field, which would result in a masquerading fault.

In an embedded system with both critical and non-critical processes, masquerading faults can occur in three different scenarios: (1) a critical process might be sending a critical message to another critical process; (2) a critical process might be sending a critical message to a non-critical process; and (3) a non-critical process might be sending a non-critical message to a critical process. Because critical processes are trusted to work correctly, critical messages are assumed to have correct information, and messages from non-critical processes are suspect. Safety problems due to masquerading can thus occur if a non-critical process sends one of the first two types of messages (i.e., if a non-critical process sends a message falsified to appear to be a critical message). An additional situation that is sometimes of concern is if a critical node suffers a hardware defect that causes it to masquerade as a different critical node, resulting in a failure of fault containment strategies (designs typically assume not only that faults are detected in critical nodes, but also that they are attributed to the correct critical node).

3 PROTECTION LEVELS

Once the fault model has been defined, an appropriate level of masquerading fault detection can be implemented based on the needs and constraints of the application. As with any engineering design, this requires tradeoffs in cost, complexity and benefits.

3.1 *Level 0 - Network protection only*

The first, baseline, level of protection is to rely solely on network error checking. Most networks provide some mechanism to detect network errors. Ethernet and the Controller Area Network (CAN), for example, both use CRCs.

The problem with relying on the network-level data integrity checks is that they only check for errors that occur at the network link level. Errors due to software defects or some hardware defects in the network interface are not detected. In addition, these detection techniques may not be very effective at detecting errors caused by routers and other networking equipment on multi-hop networks. For example, Stone and Partridge (2000) found high failure rates for the TCP checksum, even when combined with the Ethernet CRC.

Though relatively effective at preventing random bit errors, Level 0 remains vulnerable to defects in networking hardware that cause undetected message errors, software defects that result in masquerading by critical and non-critical processes, and malicious attacks. In terms of bandwidth and processing resources, this level requires no additional cost because it is already built into most networks.

3.2 *Level 1- Application CRC*

The next step in assuring message integrity is to apply an application-level CRC to the data and to include it in the message body that is transmitted on the network. Stone and Partridge (2000) strongly recommend using an application-level CRC to help detect transmission errors missed by the network checks due to defects in routers and other networking equipment.

It might be the case that some or all of the processes within a system use the same application-level CRC. If non-critical processes use the same application-level CRC as critical processes, then the application CRC provides no protection against masquerading by non-critical processes.

Level 1 provides additional protection against defects in networking hardware that cause undetected message errors. However, it does not protect against critical message sources that falsify message source information due to faults, defects that result in masquerading by critical and non-critical processes, and malicious attacks.

With respect to resource costs, Level 1 requires some additional bandwidth and processing resources, but not many. For example, on a CAN network a 16-bit application CRC requires two of the eight available data bytes and an additional few instructions per data bit of CRC. In critical systems application-level CRCs are not uncommon.

3.3 *Level 2 - Application CRC with secret polynomial/seed (symmetric)*

The application-level CRC in Level 1 may be converted from a simple data integrity check into a lightweight digital signature by using different CRC polynomials for different classes of messages. In this scheme, there are three separate CRC polynomials used: one for critical messages sent between critical processes, one for non-critical messages sent by the critical processes to non-critical processes, and one for messages sent by the non-critical processes. It is important to select "good" polynomials with an appropriate Hamming Distance for the lengths of messages being sent, of course (Siewiorek and Swarz, 1992).

The Level 2 approach is a "lightweight," symmetric digital signature in which the secret key is the CRC polynomial. It is symmetric because both the sender and receiver of a message need to know the same key, and must use it to sign messages (by adding an application-level CRC using a specific polynomial), and verify signatures (by computing the application-level CRC using an appropriate polynomial based on the purported message source and comparing it to the frame check sequence (FCS) field of the message actually sent).

A straightforward implementation involves using a different secret polynomial for each class of message: CRC_1 for critical to critical messages; CRC_2 for critical to non-critical messages; and CRC_3 for non-critical message senders. (Note that the case where non-critical processes omit an application-level CRC is equivalent to using a null CRC for situation CRC_3).

Use of three CRCs is required because this is a symmetric system. Thus, it is possible that any process possessing a CRC polynomial might send a message using that polynomial due to a software defect. If CRC_1 is only known to critical processes, that means it is impossible (or at least probabilistically unlikely) that a non-critical process can falsify a message that will be accepted by a critical process as having come from another critical process. In other words, CRC_1 is a secret symmetric key, and only key-holders can generate signed messages. CRC_2 is used to provide assurance that critical messages are being sent either from critical processes or non-critical processes (with software defects) that are receivers of critical messages. CRC_3 is simply an application-level CRC for non-critical messages. It might be the case that there is no point in distinguishing CRC_2 from CRC_3 depending on failure mode design assumptions, because in either case at least one non-critical process

would have access to the secret key CRC_2 for generating critical-process-originated messages.

With this scheme there is still a critical assumption being made about non-critical code. However, it is a much narrower assumption than with the Level 1 approach, and is probably justifiable for many situations. The assumption is that CRC_1 has been selected from a pool of candidate CRCs at random, and is unlikely to be used by non-critical processes on a statistical basis. (One assumes that "well known" published CRCs are omitted from the potential selection pool, of course.) For 24-bit or 32-bit CRCs this assumption is probably a good one, but there is still a finite number of "good" CRC polynomials that are significantly fewer than all possible 24-bit or 32-bit integers.

A solution that is even better for these purposes is to use a "secret seed" for a given polynomial. Conventional CRC calculations use a standardized starting value in the CRC accumulator, typically either 0 or -1. A secret seed approach uses some different starting, or "seed" value for computation of the application-level CRC that varies with the class of message. So instead of CRC_1, CRC_2 and CRC_3 for the previous discussion, the technique would involve using the same CRC with $Seed_1$, $Seed_2$, or $Seed_3$, with each seed being a different secret number. Thus, the seed value becomes the secret key for a digital signature.

Thus, the FCS of a message with a level of criticality *i* would be computed as follows. If CRC(M,S) takes a message M with an initial CRC seed value S to compute a FCS, then:

$$FCS_i = CRC(M,S_i) \quad (1)$$

Critical to critical process messages would be authenticated by having critical processes use S_1 to compute and compare the FCS field. Since no non-critical process would have knowledge of S_1, it would be, for practical purposes, impossible for non-critical processes to forge a correct FCS value corresponding to a critical message. There would still be a chance of an accidental "collision" between the FCS values for two CRCs, but this is true of cryptographically secure digital signatures as well, and can be managed by increasing the size of the FCS as required.

Combining a secret polynomial with a secret seed is possible as well, of course, but does not provide a fundamentally different capability. It is important to note that CRC-based digital signatures are readily attacked by cryptanalytic methods and are ***not secure against malicious attacks.*** However, in a cost-constrained system it might well be reasonable to assume that non-critical components will lack cryptanalytic attack capabilities, and that software defects will not result in the equivalent of cryptanalytic attacks on secret CRC polynomials or secret seeds.

Symmetric-key CRC lightweight digital signatures of Level 2 provide the same benefits as application-level CRCs of Level 1. In addition, they provide protection against non-malicious masquerading by non-critical processes that results in acceptance of fraudulent critical messages. However, Level 2 does not protect against non-malicious masquerading of critical message sources by other critical message sources due to faults, and is inadequate protection against malicious attacks. The benefit of Level 2 is that it requires no additional processing or bandwidth to upgrade from Level 1.

3.4 Level 3 -Application CRC with secret polynomial/secret seed (asymmetric)

Symmetric CRC-based signatures ensure that non-critical processes cannot send critical messages to critical processes by accident. However, a software defect could still cause a non-critical process to masquerade as a critical process sending a non-critical message. (This is true because all noncritical processes possess the symmetric key information for receiving such messages). Additionally, Level 2 assumes that all critical processes are defect-free, providing no protection against masquerading by a critical process in the event of a hardware failure or software defect.

A further level of protection can be gained by using asymmetric, lightweight authentication. In this approach every process has a secret sending key and a public receiving key. The public receiving key is known by all processes, but only the sending process knows the secret sending key. In such a scheme every process retains the public receiving keys of all processes from which it receives messages (in general, meaning it has the public keys of all the processes). But because each process keeps its transmission key secret, it is impossible for one process to masquerade as another.

Because embedded systems tend to use broadcast messages heavily, an implementation of full public-key encryption is impractical, so the method proposed here is tailored to a broadcast environment. Additionally, CRC-based authentication is used which is of course ***not secure against a cryptanalytic attack.***

One way to implement a private/public signature scheme is the following, using secret polynomials. This method may also be used in addition to the use of distinct CRCs or seeds for FCS computation as outlined in Level 2. If desired for cost and simplicity reasons, all non-critical to non-critical messaging can

use a single standard polynomial, and only critical message sources need use the private/public key approach.

Each critical process has two CRC polynomials: CRC_1 and CRC_2. CRC_1 is a publicly known polynomial, whereas CRC_2 is a secret private polynomial. Every CRC_1 in the system is distinct per process. Every CRC_2 is the inverse of the corresponding CRC_1. Thus, the secrecy of CRC_2 depends on there being no code to compute an inverse polynomial in the system. Because computing inverse polynomials is performed using a bit-reverse operation (with adjustments to account for an implicit 1 bit within the polynomial in most representations), the validity of the assumption of secrecy is one that must be made in the context of a particular system design. However, computing inverse polynomials off-line and putting them in as constants within the system code avoids the presence of inverse polynomial code, and might well be a reasonable approach for systems that cannot afford the cost of full-strength cryptography. (The creation of stronger, but efficient, methods for asymmetric signatures is an open area for future research.)

A sending process S appends a signature X to a critical message M and its FCS field (X is not included in the FCS computation), where "|" denotes concatenation:

$$M \mid FCS \mid X \tag{2}$$

where:

$$X = CRC_2(FCS) \tag{3}$$

Receiving processes then verify the authenticity of the transmission by ensuring that:

$$FCS = CRC_1(X) \tag{4}$$

What this is doing is "rolling back" the FCS using an inverse CRC, CRC_2, to compute a signature that, when rolled forward through CRC_1, will yield the FCS. Because only the sending process knows the inverse CRC for its public CRC, no other process can forge messages.

This method protects against software and hardware defects that cause a process to send a message that should not be sent (e.g., forged source field or incorrect message identifier/type information).

This method is vulnerable to the following: malicious attacks using cryptanalysis (even without knowledge of the public CRC polynomial); software defects involving CRC code that computes CRCs "backwards" from the critical CRC computation (e.g., right-to-left CRC computations when the critical code is using a left-to-right shift-and-xor computation); and software defects in critical or non-critical software that compute the bit-reverse of a public polynomial and then use that as the basis for signing a message. While some of these defects could probably happen in real systems, the specificity of the defects required would seem to provide a higher degree of assurance than not using such a technique. As stated previously, this is an example of a simple lightweight signature technique; it is possible that future research will yield even better approaches to fill this niche in the design space.

If the system is originally designed at Level 1 or Level 2 with an application CRC, then there is an additional cost to compute and transmit the signature X. In a CAN network with a 16-bit signature X, this would be an additional handful of instructions per CRC bit and two bytes of the remaining six available data bytes.

3.5 Level 4: Symmetric cryptography

Levels 1 through 3 all use some form of CRC to detect masquerading errors due to defects in the non-critical software, and provide no credible protection against malicious faults. The next higher level of protection can be achieved through the use of cryptographically secure digital signatures. Although designed primarily for malicious attacks, such digital signatures can also prevent defective non-critical software components from forging critical messages. This can be accomplished via use of a Message Authentication Code (MAC), which is a keyed one-way hash function. A detailed description of MACs appears in Section 18.14 of Schneier (1996).

Symmetric digital signatures must be sufficiently long to preclude successful malicious attacks via cryptanalysis or brute force guessing. Additionally, they take significant computational capability beyond the means of many embedded systems. However, a symmetric key approach is secure against malicious attacks unless the attacker compromises a node possessing a secret key. In the case that the attacker compromised a critical code, it would be possible to maliciously forge a message that apparently originated in any node in the system. Malicious attacks aside, a Level 4 approach has the same strengths and weaknesses as a Level 2 approach in that it is a similar general signature method, but using strong cryptography.

3.6 Level 5: Public-key digital signatures

Level 4 protection was analogous to Level 2 protection, but used cryptographically secure symmetric digital signatures. Level 5 is, in turn, generally similar to Level 3 CRC lightweight public key signatures, but uses cryptographically secure signature algorithms. Various public-key digital signature algorithms are described in Section 20 of Schneier (1996).

A Level 5 approach provides protection from forgery of message sources to the limits of the cryptographic strength of the digital signature scheme used. Moreover, if a node is compromised by malicious attack, forgery of messages can only be accomplished with compromised node(s) as originators, because each node has its own distinct secret signature key. However, public-key methods have longer signatures and are much slower than symmetric cryptography (Menezes *et al.*, 1997).

3.7 Tradeoffs

Each of these methods provides a certain level of fault protection; however, they each have a commensurate cost. The developers must decide what protection is required to attain safe operation, and adjust system design decisions on how much safety critical operation to delegate to computers based on budget available to provide protection against realistic masquerading threats. For example, a system with a fault model that includes software defect masquerade faults but excludes malicious attacks might chose Level 2 or Level 3.

An additional burden that must be assumed when using any masquerading detection technique is that of cryptographic key management. Any technique discussed assumes that only a certain set of nodes have access to secret keys. This restricted access results in significant complications in configuration management, deployment, and maintenance, especially when insider attacks are considered a possibility. (As a trivial example, every time a disgruntled employee leaves a company, it is advisable to change all cryptographic keys that the employee might have had access to if attacks by that employee are a substantive threat.)

Figure 1 shows all of the levels, in order of effectiveness. In general, the stronger the protection, the more expensive the method. Levels 3 and 4 have a partial ordering, because the protection of Level 3 (asymmetric secret CRC) might be more useful than the protection afforded by Level 4 (symmetric secure digital signature), depending on whether malicious attacks are a part of anticipated threats. However, it is expected that CRC-based signatures will be substantially less expensive to implement than cryptographically secure digital signatures.

4 CONCLUSIONS

This paper presents six levels of fault detection techniques that can be deployed against the possibility of masquerading faults on shared critical/non-critical fieldbuses. Level 0 (network-provided protection) provides no protection beyond what is included in the network protocol. For level 1 (published CRC), the application must be modified to apply the application-level CRC before sending messages on the network, and after messages have been received. Once an application-level CRC is present in the code, the polynomial or seed value used in the calculation can be changed to achieve Level 2 (symmetric secret polynomial/seed) protection. A novel Level 3 (asymmetric secret polynomial/seed) approach is proposed to provide very lightweight digital signatures with a public key flavor that are suitable for broadcast bus applications, but that further assume malicious faults are not a threat. Levels 4 and 5 complete the taxonomy and consist of using well known cryptographically secure approaches to guard against malicious masquerading faults.

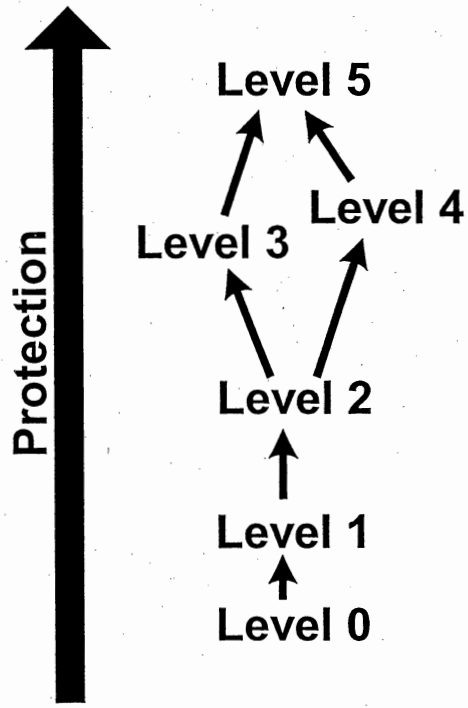

Figure 1. Masquerading fault protection levels

Typical fieldbus systems today operate at Levels 0 and 1, and are not secure against masquerading faults. It might be attractive in some applications to upgrade to a Level 2 or Level 3 capability to improve resistance to non-malicious software defect masquerade faults without having to resort to the complexity and expense of cryptographically secure Level 4 or Level 5 approach.

5 ACKNOWLEDGMENTS

This work is supported in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University, Bombardier Transportation, and by the Pennsylvania Infrastructure Technology Alliance. The authors would also like to thank Bob DiSilvestro for his loyal support.

6 REFERENCES

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

IEEE (1998). IEEE Standard for Software Quality Assurance Plans, IEEE Std 730-1998.

Menezes, A. J., P.C. Van Oorschot, and Scott A. Vanstone (1997). *Handbook of Applied Cryptography*. CRC Press LLC, Boca Raton.

Morris, J. and P. Koopman (June 2003). Software Defect Masquerade Faults in Distributed Embedded Systems. *IEEE Proceedings of the International Conference on Dependable Systems and Networks Fast Abs*.

Schneier, B. (1996). *Applied Cryptography*. second edition, John Wiley & Sons, New York.

Siewiorek, D. P. and R. S. Swarz (1992). *Reliable Computer Systems Design and Evaluation*. Second Edition. Digital Press, Bedford, MA.

Stone, J. and C. Partridge (2000). When the CRC and TCP Checksum Disagree. *ACM SIGCOMM Computer Communication Review: Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 309-319.

5th IFAC International Conference on
Fieldbus Systems and their Applications

FeT'2003

Aveiro, Portugal, July 7-8, 2003
http://www.det.ua.pt/eventos/fet2003

Proceedings Preprints

Sponsored by:

IFAC

International Federation of
Automatic Control
TC on Components and Instruments

Hosted by:

Universidade de Aveiro
Portugal

Organized by

ieeta instituto de engenharia electrónica e telemática de aveiro

Co-sponsored by

IFAC Technical Committees:

Computers for Control,
Manufacturing Plant Control.

APCA Portuguese National Member Organization

Organized jointly with SICICA 2003
http://www.det.ua.pt/eventos/sicica2003

Supported by

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA E DO ENSINO SUPERIOR Portugal

Cover designed by Sérgio Cabaço
Special thanks to António Neves, author of the painting illustrated in the cover