An efficient approach to tolerate attackers in fault-tolerant systems

Johannes Formann
Universität Duisburg-Essen, Essen
formann@dc.uni-due.de

Abstract: Malicious attackers can cause severe damage (financially or to the environment) if they gain control of safety-relevant systems. This paper shows why the traditional disjoint treatment of security and fault tolerance has weaknesses if the attacker gains access to the fault tolerant system and how an integrated approach that utilize existing fault tolerance techniques could be an effective security mechanism. An efficient integrated safety and security approach is presented for fault tolerant systems, which achieves protection against attacks via the network by forming a logically isolated (sub-) network which is resilient against a bug in the codebase. Isolation is obtained by diverse design of a general reusable (software and/or hardware) component that prevents any unauthorized message transfer towards the secured application program. Messages from other compromised nodes are tolerated utilizing existing majority voting mechanism.

1 Introduction

Modern distributed safety-relevant systems utilize different techniques to ensure fault tolerance. They usually assume that their communication network is sufficiently isolated, and thus protected against launching of malicious attacks. However in various domains the communication systems are more and more opened to external networks. In industrial automation the so-called horizontal and vertical integration opens the network to other automation systems as well as to administration and office networks. Moreover, studies show [BE04, SM07] that these networks are often not very secure.
In some domains, e. g. automotive or energy distribution, modern networks are designed to have communication with the outside world (e. g. Car-to-Car or Car-to-Infrastructure [HKD08, SR10]). Some voices say that the increasing connectivity could enable attacks even in safety-relevant systems, while others still assume the network to be kept isolated through out its entire lifespan.
The naive approach of protecting the network by a firewall has different drawbacks. One of the main drawbacks lies in the fact that the complete communication has to pass the firewall. However, as described in [BE04], there are often many communication channels that must be secured. Another drawback: the firewall must be able to distinguish between valid and malicious traffic. To ensure this in the case an attacker has gained control of a legitimate sender or in the case of masquerade the firewall needs to implement checks for the payload. This means a firewall has to be designed individually for the specific applications. This results in high costs (no “component of the shelf”) and increases the

probability that the firewall itself might have a bug that enables an attacker to access the internal network. In this paper we assume that it is possible for an attacker to gain access to the network of a safety-relevant system, since there are many examples (see appendix A.1) that proves this is a valid assumption.
We will show an approach to logically isolate the safety-relevant system and make it resilient against an attacker which has access to a remotely exploitable vulnerability, even if the vulnerability is in the code handling the security mechanism. The approach is based on diversely developed communication hypervisors and majority voting. The solution is completely implemented in an application-independent component. Thus it is reusable for any application.

2 Why redundant replicas do not solve the problem

At a first glance an attacker with knowledge of just a single exploitable vulnerability in the software is not a fundamentally different threat to a system compared to other threats from the environment (radiation, thermal stress, ...). Both can result in erroneous internal states [Jon99] of the program, which in turn can result in wrong behavior (e.g. malicious byzantine faults). These effects are regularly considered in fault-tolerant systems [Ech90]. However an attacker can be even worse since his behavior might not match the assumptions typically made for the tolerance of technical faults. Examples are addressed in the following sections.

2.1 Number of affected nodes

In most systems redundancy for fault tolerance is accomplished by using n replicas of a node. Usually n is $2f + 1$ (in case of majority voting) or $3f + 1$ (in case of Byzantine agreement) assuming up to f faulty nodes. A bug is a systematic fault [KA83, CMSB08, KL86] that has been introduced while programming or setting up the system. Normally the program is replicated for all redundant nodes. Consequently, nothing prevents the attacker to take control of all n replicas. This clearly violates the assumption of a maximum of f faulty nodes. Since the attacker can control the majority (or all) nodes, he can control the complete system. In [GÖ3] an approach has been proposed to model the security and safety over time in the presence of an attacker.
An exception of this rule is the case where redundant nodes own independently/diversely developed program versions rather than instances of the very same program. When the programs in the redundant nodes differ from each other, a common design fault is still possible, but not likely to occur [KA83, KL86, ALS88].

2.2 Assumptions regarding the behavior

In many systems there are not only limitations regarding the number of tolerated faults, there are also assumptions with respect to the behavior a faulty node can exhibit. A frequently used assumption says that faults of nodes are stochastically independent [Sch90]. If the behavior is “worse” than the assumed one, fault tolerance is not guaranteed. If these assumptions do not include malicious Byzantine faults (if technically possible) and the possibility that all faulty nodes cooperate, the assumptions are likely to be violated by a malicious attacker who controls f faulty nodes.

2.3 Resilient communication system

A distributed fault-tolerant system usually depends on its communication infrastructure. Many systems are designed to tolerate a loss of one communication channel (e.g. FlexRay [Fle05] configured to use both buses for redundancy) but usually the fault model assumes one or more random errors in the communication system, not a malicious attacker. Most communication systems have some kind of communication controllers (FlexRay, CAN [Bos91]) which are usually configured by the node they are attached to. In this case a malicious attacker can simply reconfigure its communication controller to have the biggest possible impact on the communication system. Very simple communication systems like LIN [LC10] can be (almost completely) implemented in software. Such communication systems are designated for disturbing the communication by an attacker. It has been shown for different bus systems, that effective denial of service attacks are possible [WWW07, JM06] by capturing only a single node.

3 Definitions

3.1 Program

A program provides a function in a system and has clearly defined interfaces to the outside world. In this Paper the interface is assumed to consist only of message transmission over the communication system (to exchange information with other programs in different nodes) and of interaction with the outside world by simple[1] IO-Interfaces. Since simple IO does not provide any means to transport a specially crafted payload that can be used to exploit vulnerabilities in the program and thereby modify the behavior, they do not need any further protection.
A program can be designed to work in a fault-tolerant environment e. g. by using multiple program instances (distributed over redundant nodes), majority voting for the input, and multicasting of results to multiple instances of the program that further processes the data (also distributed over redundant nodes).
A program is assumed to be fault-free, if the majority of the program replicas work correctly, so that the output can be sent to voters, which will mask out wrong results. Typically the voters are located at the input side of the replicas of the program that processes the output data further.

[1]No communication protocol is used on top.

3.2 Node

In this paper a node is a physical entity that is at least composed of a microcontroller and an interface to the communication system. A node might only support a single specialized application (e. g. a node that is built to read a value from a sensor and then transmit the value to other nodes) or utilize some kind of operating system to support different local applications (e. g. a so-called PLC (programmable logic controller. In this work a single node is also the smallest entity an attacker could gain access to. It is assumed that access is achieved by exploiting a bug regardless if the bug is in the application or the operating system (if present). In any case we assume the attacker gains full control of the node.

3.3 System

A system delivers at least one (fault-tolerant) service to the outside world. To provide a service the system can be internally subdivided into programs. In this paper it is assumed that a system is working properly, if all programs are working correctly.
To ensure the logical isolation it is assumed, that the set of nodes that runs the programs are not used for other systems. Furthermore, it is assumed that each node could have one or more potentially exploitable vulnerabilities. In all, the system consists of different (potentially intersecting) subsets of nodes owning the same exploitable bug, which can be misused over the network to obtain control over the node. To avoid making assumptions regarding the vulnerability, we assume – as said before – that the attacker can take full control of a node if he has knowledge how to exploit a vulnerability.

3.4 Attacker

The attacker is an entity trying to modify the system behavior to operate outside the designed safe envelope. It can utilizes intelligence and external resources (e.g. computing power) to archive this goal.

3.5 Communication system

The communication system ensures that a message is delivered from the sender to the receiver. The necessary properties depend on the service that the system has to deliver. If the service has no safe state that can be reached easily, it is usually called a “fail-operational” service. That means the system must deliver the service even in the case of an error. In this case communication has to be resilient against denial of service attacks. However, if the service can reach a safe state, the system is usually designed with watchdogs, which ensure the safe state is activated in the case of a communication breakdown. In this case no protection against denial of service is required.
The attacks on the communication system itself and the respective countermeasures are not in the focus of this paper.

4 Cryptographic Island

In this chapter the new approach is presented, that leverages existing fault tolerance techniques to create an application independent framework which provides protection against an attack over an existing communication system.
The cryptographic island utilizes two basic design principles. Logical isolation through cryptographic signatures isolates the application from external threats. To avoid a common vulnerability in the code that ensures the logical isolation, a diversely developed framework/hypervisor is utilized to enforce the logical isolation. To reduce the necessary development effort the framework/hypervisor should be designed in such a way, that it has a generic interface towards the programs, so it can easily be reused.

4.1 Assumptions

The concept of the cryptographic island relies on some assumptions.

- The attacker may somehow gain access to the communication system, but the he/she has no physical access to any of the nodes of the safety-relevant system. For example the attacker might have gained control of another node (e. q. some monitoring service used for maintenance) connected to the communication system or found an unguarded connection to this network.
- The communication system provides the needed level of robustness against the attacker (see chapter 3.5).
- There is no common bug in the diverse framework variant. According to [KA83, KL86, ALS88, GV79] common bugs are very unlikely. To reduce the probability that an (unlikely) common bug can be utilized by the same exploit method to gain control of the node, different "tool chains" (i. e. programming languages, libraries, compilers) could be used. In this case there is a good chance that an exploit works only for one framework variant, and the other variants are not affected (message is detected as being erroneous) or they expose a much easier to handle fail-silent behavior and simply crash.
- There are no configuration errors. This means especially that all replicas of a process are distributed on nodes with different framework variants. Automated checks could be developed to ensure this property at system startup.
- The attacker has only knowledge of a known and limited number of vulnerabilities that he/she can use for gaining access. If there are more exploitable bugs, the other vulnerabilities are unknown to the attacker. This is the equivalent to the f fault assumption for fault tolerant systems.
  Without making further assumptions regarding the application behavior running on top of the framework, the number of vulnerabilitie sis limited to one. This assumption could be considered valid if further techniques like those presented in [SPS08, CS11] are used to detect the presence of an attacker on the compromised

node. After detection one can either direct the system into a safe state within a time period that is too short for an attacker to find a second exploitable vulnerability. Alternatively one can reconfigure the system by using spares (this means: additional diverse variants of the framework) or repairing the system in a timely matter by fixing the bug so that all assumptions are met again and the compromised nodes are no longer part of the system.
Different from the generic case, two approaches to tolerate more than one known bug are presented in chapter 4.4.

4.2 Logical isolation

As stated in the introduction, an isolated environment is a good protection against malicious attackers. Since physical isolation is not always feasible, logical isolation can be utilized. Logical isolation can be achieved by the combination of an effective protection against masquerading, and list of legitimate senders for each program instance.
Then for all incoming messages a node knows the real senders and it is able to decide whether it wants to accept a message or to drop it.
An effective protection against masquerading could be implemented in the communication system. If the communication system does not offer an effective masquerade protection (and most communication systems do not) the masquerade protection must be implemented inside the framework. Strong cryptographic signatures are an effective method to verify the identity of the sender from a message. Some approaches have already been discussed for embedded (realtime) networks [GR97, SK09, EKP$^+$07, WWP04]. The strength (usually measured in key-length) of the used cryptographic methods depends on the computing power of the attacker. Typically the computing power of the attacker has to be assumed unknown, and consequently considered to be large.

4.3 Reusable diverse framework

To protect the system, it is necessary that the application is protected against unauthorized senders, that may know a bug in the application. But the attacker might also know a bug inside the logic that checks the cryptographic signatures or the message handling before the signature check. To ensure that the system is resilient against both kind of attacks, the protection of the application can be implemented using a framework that acts as a kind of hypervisor checking every communication. To mitigate bugs in the hypervisor, it is recommended that different variants of this framework are developed using diverse programming. Thus a common vulnerability in two or more different variants of the framework is very unlikely. To create a fault-tolerant application that is also resilient against known bugs the different replicas of the same program have to be executed on top of different diverse variants of the framework.
Figure 3 in appendix A.2 show the possible locations of the framework within a node. The framework can either be purely implemented in software (see figure 3a) or comprise both software and hardware (see figure 3b).

For an example assume that programs P and Q have to be executed by three redundant replicas each. Moreover, assume that diverse variants A, B and C of the framework have been implemented. Then the replicas P_1, P_2 and P_3 of P, and Q_1, Q_2 and Q_3 of Q may be executed as follows: replicas P_1 and Q_3 on top of A, replicas P_2 and Q_2 on top of B, and replicas P_3 and Q_1 on top of C. Permutations within (P_1, P_2, P_3) or within (Q_1 Q_2, Q_3) will also work, of course. The allocation to nodes is not determined by this approach. It can be chosen according to performance considerations or to the available IO connections. Let N_1, N_2, N_3, N_4 and N_5 be a set of five nodes. Then one of many potential allocations is the following (see figure 1):

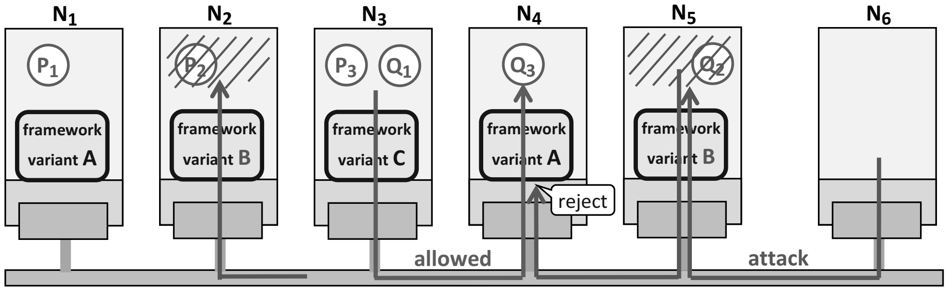


Figure 1: Diverse framework in the example system

nodeN_1 runs A and P_1 on top of it	node N_4 runs A and Q_3 on top of it
node N_2 runs B and P_2 on top of it	node N_5 runs B and Q_2 on top of it
node N_3 runs C and P_3 and Q_1 on top of it	

Any node does not run more than one variant of the framework.
If an attacker has found an exploitable vulnerability in one of the framework variants (variant B in the example depicted in figure 1), only the replicas on top of this single framework variant are affected (which means that all nodes running this particular variant can be compromised). If the degree of redundancy is sufficient, then for each process only the minority of variants is affected. Thus majority voting will decide for the correct result despite the attack. In figure 1 the majority P_1 and P_3 of process P (using variants A and C respectively) and the majority Q_1 and Q_3 of process Q exist and produce correct results despite the attack.
To ensure this isolation, messages are signed using strong cryptography by the sender. Each receiver can check the true origin of a message it obtains and, moreover, can check if the message has not been modified during transmission via the network. The framework on the receiving nodes checks for each message if the stated sender is an allowed sender and if the signature of the message is valid for the stated sender. If both checks are passed successfully the message is delivered to the program. Otherwise it is discarded. It is assumed that the information required for these checks (the necessary cryptographic certificates of the valid senders) have been distributed safely, for example by deploying during the initial system deployment. The limitation to trusted and well-known sender result in logical point-to-point connections for the programs.
If a system consists of programs each of which is made fault-tolerant by replication, and,

moreover, the replicas of a program are distributed among the nodes in such a way, that each replica utilizes a different version of the framework, then the system is safe against an attacker that has knowledge of how to exploit a single vulnerability. The attacker is unable to control the majority of the replicas.
Assuming the key distribution is done without carelessness, an attacker can't attack the application itself, since the message is dropped due to the lack of a valid signature. Therefore the only chance to influence the system is to find a bug in one of the diverse framework variants that guards the message flow. If the attacker gains access to one remotely exploitable vulnerability, he/she could take control over all nodes running the same framework variant. If the system is designed as described, this will only affect a minority of the replicas of each application. So the potential erroneous results created by the attacker-controlled nodes could be easily tolerated in the applications like any other erroneous result (e. g. by applying majority voting).
Since only a single remotely exploitable bug is assumed, the attack is unsuccessful.
The development of a framework that acts as a hypervisor for message transmissions has two main advantages over complete diverse development. The first advantage is that once this framework has been developed, it can be used for different programs and systems. This reuse significantly reduces the costs for each project and potentially the framework versions could become "components of the shelf" if the environment is standardized like the AUTOSAR [AUT10] environment for the automotive domain. The second advantage is that only a small part of the functionality must be developed in a diverse way. This functionality is not necessarily implemented in software. Some parts of the functionality (e. g. the computation of the cryptographic signature due to performance reasons) or even the full functionality can be implemented in hardware.
The application programs themselves need not be programmed diversely. This further reduces the necessary effort.

4.4 Tolerating multiple bugs

Multiple bugs can be tolerated if the framework ensures, that the attacker is unable to send an exploit to a replica of the application on a unexploited framework version. This results in two requirements:

- The attacker must be unable to send a message the the protected application on nodes with an unaffected framework version while there are still known bugs.
- The attacker must be unable to determine a input signal that is required for the application. This usually means that the attacker is not allowed to control the majority of the nodes sending the input signal.

The basic approach does not ensure this property. An attacker with knowledge of two (or more) exploitable bugs might use the first exploit to gain access to a node that sends messages to all replicas of an application. If the second exploit works against the application, the attacker gains access to all replicas of this application, since he is able to sign the messages with the legitimate key he gained by accessing the first attacked node. With access

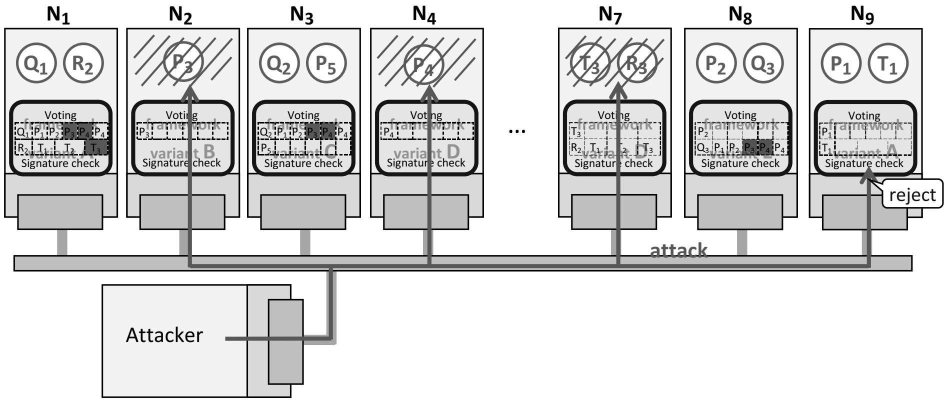


Figure 2: Example for voting inside the framework

to all replicas of a program, he could determine the output of all the replicas and therefore, independent of the number of tolerated faults, influence the system behavior.
Currently two different approaches for tolerating multiple bugs are research topics. The first approach is voting inside the framework and the second approach is usage of different framework variants on top of each other to create multiple protection layers. Possible realizations of the multi-layered approach are subject of ongoing research.

4.4.1 Voting inside the framework

To ensure, that the protected application does not receive an erroneous message from the attacker, the voting algorithm used by the fault-tolerant application can be placed inside the framework. When a message arrives the signature is checked. If the signature is valid the message is stored inside the framework. After the messages of all replica of a input program have arrived (or a timeout has expired) the framework executes a voting algorithm and delivers the result to the application. Messages from nodes controlled by the attacker are treated like all other (faulty) input messages, as long as the faulty/malicious inputs are the minority, they are discarded.
If there are $2f + 1$ replicas of the protected application on $2f + 1$ framework variants (with f being the number of faults including the exploitable bugs known to the attacker) the attacker could not attack the application, since he cannot affect the majority of used framework variants. If the used input messages are protected the same way ($2f+1$ replicas of the application generating the input on $2f + 1$ framework variants) the attacker could not control the majority too. Consequently the attacker can only control a minority of input messages, that are being discarded during the majority voting inside the fault-free framework versions.
In figure 2 an example is shown which shows an excerpt from a larger system with four applications (P, Q, R, T). The replicas of application P generate the input for the replicas of application Q and the replicas of application T generate the input for the replicas of application R. The inputs for P and T are not shown in this figure and the corresponding

fields in the tables are left blank. The applications P and Q tolerate two bugs or faults (resulting in five replicas running on top of five framework versions) and the applications R and T tolerate one bug/fault (resulting in three replicas running on top of three different framework versions).
The voting have to be programmed diversely as the rest of the framework, to reduce the risk that an attacker may find a common bug inside the majority voter. To enable reusability and keep the message handling logic small the voting algorithm inside the framework should be sufficiently simple (e.g. majority voting).
This approach has the advantage, that it is possible to tolerate more than one known bug, but it sets a few constraints for the applications. The application have to be designed to delegate voting to the framework, therefore the framework can not designed as a "drop-in-replacement" for existing libraries without protection. The application is restricted to simple (majority) voting algorithm. Sophisticated algorithms based on system behavior models would contradict the reusability and increase the codebase which increases the likelihood of bugs.

4.4.2 Multiple protection layer

Multiple protection layers stacked on top of each other have basically two advantages compared to the voting inside the framework:

- There are no assumptions regarding the program behavior. The program can use as complex voting algorithms as desired and the framework stack could be designed to be a "drop-in" replacement for the existing message sending and receiving API.
- The number of tolerated bugs could differ from the number of tolerated faults.

The main problem with this approach is caused by the fact, that the layers have to be independent from each other. If the attacker finds a bug in a protection layer, he/she must not be able to manipulate the other protection layers. This clearly violates the assumption in chapter 3.2 that an attacker has access to all software running on a node, once he gained access. There is ongoing research for system designs, where a different assumption for the access after an exploit could be justified.

5 Conclusion

Malicious attackers are a real world problem when designing modern safety-relevant systems. It attracts more and more attention of designers who are confronted with both safety and security issues. If the classical approach of full diversely developed software is not feasible, the new approach of forming cryptographic island provides a solution for logical isolation of the application programs against external attackers. The approach achieves a good tradeoff between security and necessary development effort leveraging the existing mechanism for fault tolerance and requires only a small reusable framework.
To make this approach usable for a broader set of use cases a special focus is set on the admission of external input from (potentially) untrusted systems via the communication network. This is a necessity for all scenarios where an external input (e. g. production

planning systems of a chemical plant, infrastructure-to-car communication in the automotive application) must be able to modify the behavior of the system in a “safe envelope” without affecting the safety and security goals.

References

[ALS88] A Avizienis, MR Lyu, and W Schutz. In search of effective diversity: a six-language study of fault-tolerant flight control software. In *The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, volume 1, pages 15–22. IEEE Comput. Soc. Press, 1988.

[AUT10] AUTOSAR. Specification of SW-C End-to-End Communication Protection Library, October 2010.

[BE04] Eric Byres and P Eng. The Myths and Facts behind Cyber Security Risks for Industrial Control Systems The BCIT Industrial Security Incident Database (ISID). In *VDE Kongress*, pages 1–6, 2004.

[Bos91] R Bosch. CAN specification version 2.0, September 1991.

[BS10] WILLIAM J. BROAD and DAVID E. SANGER. Worm Was Perfect for Sabotaging Centrifuges, November 2010.

[CMSB08] Byung-gon Chun, Petros Maniatis, Scott Shenker, and U C Berkeley. Diverse Replication for Single-Machine Byzantine-Fault Tolerance Challenges in a BFT Server. In *ATC’08 USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 287–292, Berkeley, CA, 2008. USENIX Association.

[CS11] Ang Cui and S Stolfo. Defending embedded systems with software symbiotes. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 358–377, Menlo Park, CA, USA, September 2011. Springer Berlin Heidelberg.

[Ech90] Klaus Echtle. *Fehlertoleranzverfahren*. Studienreihe Informatik. Springer, 1990.

[EKP$^+$07] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A Survey of Lightweight-Cryptography Implementations. *IEEE Design & Test of Computers*, 24(6):522–533, November 2007.

[Fil11] Jonathan Fildes. Stuxnet virus targets and spread revealed, November 2011.

[Fle05] FlexRay Consortium. FlexRay Communications System - Protocol Specification Version 2.1, December 2005.

[FMC11] Nicolas Falliere, LO Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security . . .* , 4(February):1–69, 2011.

[GÖ3] FC Gärtner. Byzantine Failures and Security: Arbitrary is not (always) Random. In Rüdiger Grimm, Hubert B. Keller, and Kai Rannenberg, editors, *GI Jahrestagung (Schwerpunkt "Sicherheit - Schutz und Zuverlässigkeit")*, pages 127–138. Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, GI, 2003.

[GR97] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In *Advances in Cryptology - CRYPTO'97*, pages pp 180–197, Santa Barbara, California, USA, 1997. Springer Berlin Heidelberg.

[GV79] L Gmeiner and U Voges. Software diversity in reactor protection systems: An experiment. In R. Lauber, editor, *Safety of Computer Control Systems (Proceedings of IFAC Workshop on safety of computer control systems)*, pages 75–79. Pergamon Press, 1979.

[HKD08] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security threats to automotive CAN networks - practical examples and selected short-term countermeasures. In *Computer Safety, Reliability, and Security*, pages 235–248. Springer Verlag Berlin-Heidelberg, 2008.

[IC12] ICS-CERT. KEY MANAGEMENT ERRORS IN RUGGEDCOM'S RUGGED OPERATING SYSTEM, August 2012.

[Jc 12] Jc (JC CREW). Undocumented Backdoor Access to RuggedCom Devices, April 2012.

[JM06] Michael Jenkins and SM Mahmud. Security needs for the future intelligent vehicles. In *2006 SAE World Congress*, number 724, Detroit, MI, USA, 2006.

[Jon99] Erland Jonsson. On the functional relation between security and dependability impairments. *1999 workshop on New security*, pages 104–111, 1999.

[KA83] JPJ Kelly and A Avizienis. A specification-oriented multi-version software experiment. *Digest of Papers FTCS-13: Thirteenth International ...*, pages 120–126, 1983.

[KCR+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[Lan11] Ralph Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy Magazine*, 9(3):49–51, May 2011.

[LC10] LIN-Consortium. LIN specification package (Rev 2.2A), 2010.

[Sch90] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[SK09] Christopher Szilagyi and Philip Koopman. Flexible multicast authentication for time-triggered embedded control network applications. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 165–174. IEEE, June 2009.

[SM07] Jill Slay and Michael Miller. Lessons learned from the maroochy water breach. *Critical Infrastructure Protection*, 253:73–82, 2007.

[SPS08] A Shah, Adrian Perrig, and Bruno Sinopoli. Mechanisms to provide integrity in SCADA and PCS devices *. In *International Workshop on Cyber-Physical Systems - Challenges and Applications (CPS-CA '08)*, Santorini, Greece, June 2008.

[SR10] Hendrik Schweppe and Yves Roudier. Security issues in vehicular systems: threats, emerging solutions and standards. *Presented at SAR-SSI*, pages 1–5, 2010.

[Tes13] Hugo (n.runs AG) Teso. Aircraft Hacking - Practical Aero Series, April 2013.

[WWP04] Marko Wolf, André Weimerskirch, and Christof Paar. Security in automotive bus systems. In *Proceedings of the Workshop on Embedded Security in Cars (escar)'04*, pages 1–13, 2004.

[WWW07] Marko Wolf, André Weimerskirch, and Thomas Wollinger. State of the Art: Embedding Security in Vehicles. *EURASIP Journal on Embedded Systems*, 2007:1–16, 2007.

[ZK13] Zeljka Zorz and Berislav Kucan. Hijacking airplanes with an Android phone, April 2013.

A Appendix

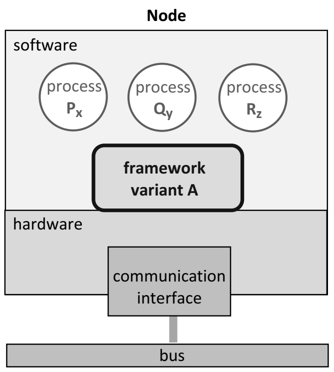
A.1 Examples of attacks to safety relevant systems

The following examples show, that the risk of unauthorized access is real and should be dealt with when designing a system.

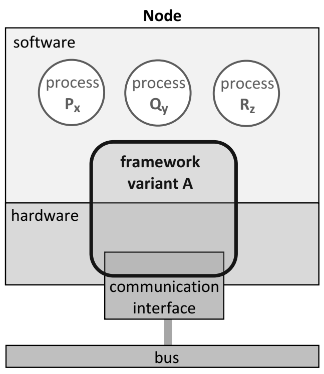
- The StuxNet attack [FMC11, Lan11] on the uranium enrichment plant in Iran is well-known and got even attention in the mainstream press [Fil11, BS10]. It utilizes different exploits to gain access to the controllers and to manipulate them without triggering the emergency shutdown system.
  This attack demonstrates that even highly secured systems that have probably physically isolated networks are vulnerable against attacks.

- A recent talk [Tes13, ZK13] at the Hack In The Box Conference in Amsterdam has demonstrated a successful attack on a (specific model of an) autopilot for a plane from the ground utilizing the wireless communication systems ADS-B and ACARS that are installed in nearly all commercial planes and are active during the flight. The attack enabled modification of the plane's course and the flight level (the altitude). This demonstrates that even in industries that obey very high safety standards security problems may occur.
- An investigation in the year 2010 of a 2009 car model [KCR+10] showed that even without the presence of Car2Car or mobile internet access an attacker could manipulate the car (including acceleration, breaking, disabling the brakes) by exploiting different vulnerabilities.
  It has been shown that the starting point of the attack could be any device connected to the internal networks. In this study for most of the experiments a small WiFi enabled OBD-Plug in the engine compartment has been used. There was also a proof of concept, where the radio, which is connected to internal bus systems to display the current song in the dashboard, was the entry point. The attack was possible since the radio had a vulnerability that could be exploited with a modified audio CD.
- A former contractor used the wireless network in a water treatment plant to gain access. Over several weeks he pumped thousands cubic meters of untreated wastewater into a river [SM07].
- In the operating system of Programmable Logic Controllers [IC12, Jc 12] remotely exploitable bugs have been found, that are independent from the used application.

A.2 Illustration



(a) Implemented in software



(b) Implemented in hard- and software

Figure 3: Framework impkementations