TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTERARBEIT

Security mechanisms for low-end embedded systems

A Proof-of-Concept for Home and Building Automation

Ausgeführt am Institut für

Rechnergestützte Automation
Arbeitsbereich Automatisierungssysteme

der Technischen Universität Wien

unter der Anleitung von

ao. Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Kastner
und
Mag. Dipl.-Ing. Fritz Praus

durch

Thomas Flanitzer

Spitalgasse 5/3/7
2432 Schwadorf

Wien, 1. Juli 2008

Kurzfassung

Die Verwendung von Embedded Systems (ES) in unserem täglichen Leben nimmt ständig zu. Deswegen wird es immer wichtiger, solchen Systemen ein gewisses Vertrauen entgegenbringen zu können, was besondere Beachtung von Security-Aspekten erfordert. Allerdings ist es aufgrund von begrenzten Ressourcen und rauen Umfeldbedingungen im Allgemeinen schwierig, Security-Mechanismen auf ES anzuwenden.

Heim- und Gebäudeautomationssysteme (HGA) verwenden Kontrollnetzwerke, um auf Geräte innerhalb eines Gebäudes zuzugreifen und um die Effizienz, Flexibilität und Steuerung des Gebäudes zu verbessern. Typische Anwendungsfälle beinhalten die Steuerung von Beleuchtung, Heizung, Belüftung und Klimatisierung. In solchen Systemen kann eine spezielle Geräteklasse ausgemacht werden, welche die typischem Charakteristiken von low-end ES aufweisen: Sensoren, Aktuatoren and Controller (SACs). SACs spielen eine wichtige Rolle in HGA-systemen und werden oft in großer Zahl verwendet.

Das Ziel dieser Masterarbeit ist die Bereitstellung von allgemeiner Security in low-end ES während es trotzdem möglich sein soll, beliebige und ungeprüfte (und möglicherweise fehlerhafte und bösartige) Anwendungsprogramme auszuführen. Das System und dessen Umfeld sollen vor Security-Angriffen geschützt werden. Eine Vielzahl existierender Mechanismen zur Verbesserung von Security auf der Ebende von Anwendungen werden geprüft und in ihrer Anwendbarkeit auf low-end ES evaluiert. Geeignete Mechanismen werden ausgewählt und zu einer Konzeptarchitektur kombiniert, welche als effiziente und sichere Lösung vorgestellt wird, um Resistenz gegen jede Art von Software-Attacken zu gewährleisten. Die Umsetzbarkeit und Effektivität der Architektur wird anhand einer Proof-of-Concept (PoC) Implementierung demonstriert, welche im Rahmen der Masterarbeit entwickelt wurde. Der PoC ist dem funktionellen Profil eines SAC-Gerätes sehr ähnlich. Eine Testumgebung, bestehend aus einem HGA Netzwerk, wurde aufgebaut und mehrere Beispiele und Tests wurden durchgeführt. In diesen Tests lieferte der PoC einige interessante und vielversprechende Ergebnisse, welche das Potential der vorgestellten Architektur untermauern.

Abstract

The use of embedded systems (ES) in our daily lives is growing. Therefore, it is becoming important to be able to put a certain amount of trust on them which requires strong consideration of security aspects. However, due to limited resources and harsh environmental conditions, it is generally difficult to apply security mechanisms to ES.

Home and Building Automation (HBA) systems use control networks for accessing devices inside buildings to improve the building's efficiency, flexibility and control. Typical applications include the control of lighting, heating, ventilation and air-conditioning. In such systems, a special device class called Sensors, Actuators and Controllers (SACs) can be be identified to have the typical characteristics of low-end ES. SACs play an important role in HBA systems and are often used in large numbers.

The goal of this thesis is to provide general security in low-end ES while still allowing to run arbitrary and uninspected (and possible erroneous and malicious) application programs. The system as well as its environment should be protected against security attacks. Various existing mechanisms for improving application level security are reviewed and evaluated in terms of applicability to low-end embedded systems. Suitable mechanisms are selected and combined into a concept architecture which is introduced as an efficient and secure solution for providing resistance to any kind of software-based attack. The feasibility and effectiveness of the architecture is demonstrated with a Proof-of-Concept (PoC) implementation which was developed as part of the thesis. The PoC closely resembles the typical functional profile of a SAC device. A test environment, consisting of an HBA network, was built and several examples and tests were run. In these tests, the PoC delivered some interesting and promising results which support the potential of the proposed architecture.

Danksagungen

Ich möchte mich an dieser Stelle bei allen Menschen bedanken, die mich während meines Studiums und insbesondere während dem Schreiben meiner Diplomarbeit unterstützt haben.

Besonderer Dank gilt meiner Mutter, Eva Flanitzer, die mir für den Großteil meiner Studienzeit ein relativ sorgenfreies Leben ermöglicht hat und ohne deren Unterstützung ich es auf dem Weg zum Abschluss wesentlich schwerer gehabt hätte. Danke Mama!

Vielen Dank auch an die Mitarbeiter des Arbeitsbereichs Automatisierungssysteme für deren fachliche Unterstützung und freundliche Aufnahme, insbesondere an meinen Betreuer Fritz Praus, Prof. Wolfgang Kastner, Wolfgang Granzer und Christian Reinisch, welche in entspannter Atmosphäre hervorragende Arbeit leisten.

Nicht zuletzt möchte ich mich bei Daniela für ihr Verständnis und den Beistand, den sie mir in schwierigen Zeiten geleistet hat, bedanken.

Contents

1 Introduction

The use of embedded systems (ES) in our daily lives is growing. They are integrated into more and more devices from mobile phones to refrigerators. The functionality of such systems is also advancing, especially when it comes to interconnection and communication. A lot of tasks are overtaken by ES, including important and critical ones. Therefore we have to be able to put a certain amount of trust on embedded systems which requires strong consideration of security aspects. The systems have to be resistant against misuse and attacks.

Providing security in ES is particularly challenging. Besides common software security threats, certain constraints complicate the task, like low resources and processing power. This makes the implementation of traditional security mechanisms difficult, as they usually have high demands in these areas. ES also often have to work in a harsh and untrusted environment, which allows attackers to penetrate them physically and by analyzing side channels.

Home and Building automation (HBA) systems use control networks for accessing devices inside buildings to improve the building‘s efficiency, flexibility and comfort. Typical applications are the control of lighting and shading, heating, ventilation and air conditioning (HVAC) as well as security and safety relevant functions. The physical interaction with the environment is done by a device class called Sensors, Actuators and Controllers (SACs). These devices typically have to be as low cost as possible, since they are often used extensively in a building's installation. Additional requirements include low power consumption and fast response time. These requirements typically result in limited available processing power and resources which makes the use of security mechanisms on such systems particularly difficult.

In almost the same manner as SACs are important and common in HBA system installations, they are threatened by security attacks. They need to be physically placed into an environment that often cannot be secured against malicious intervention (for example consider a light switch in an open place). Therefore it has to be assumed that an attacker has full physical control over the device, making it easier to perform complex attacks and to gain secret information possibly stored in the device. This harsh environmental conditions together with the limited resources make it particularly challenging to secure SACs and significant simplification and adaption is needed to be able to apply traditional security concepts.

The goal of this thesis is to provide general security in low-end embedded systems like the mentioned SACs while still allowing to run arbitrary and uninspected (and possibly erroneous and malicious) application programs. In this context, application programs refer to programs designed to perform a specific function and

which run on top of a system software. The system running the applications as well as its environment should be protected against security attacks. The assumption is that an attacker who has full control over the application program tries to cause harm which is to be prohibited. Furthermore, the solution should be applicable to devices consisting of standard components and not require any special hardware modifications.

The main difficulty in accomplishing this goal is to utilize security mechanisms using only very limited resources and processing power. Although a certain performance sacrifice may be made, the solution should still be efficient and flexible enough to allow its deployment in a wide range of applications. Additionally, programming such a system should be simple, the use of any special techniques should not be required. The programmer should be relieved of any detail knowledge of the underlying security mechanisms and be able to concentrate on the desired system functionality.

As a preparation of achieving the stated goal, security threats are classified and analyzed. They are separated into software, physical and side-channel attacks. The main focus is laid on protection against software attacks, while physical and side-channel attacks are only considered to a little extent.

Various existing mechanisms for improving application level security are reviewed. They include static and dynamic software techniques while some hardware based techniques are also considered. To provide an overview, the techniques are categorized into several general approaches. The approaches are evaluated in terms of applicability to low-end embedded systems by analyzing their capabilities and requirements to work efficiently. Conclusions are drawn which approaches could be applied to such systems.

A concept architecture reasonably combining a selection of the described mechanisms is introduced as an efficient and secure solution for low-end embedded systems. It combines Static Code Analysis, Anomaly-based Intrusion Detection and Sandboxing which in this combination can provide security against any kind of software-based attacks. Additionally, detailed instructions are given about how such a system may be implemented. The suggested architecture enables utilization of the same system software on different hardware platforms and therefore high portability of application programs.

The feasibility and effectiveness of the architecture is demonstrated with a Proof-of-Concept (PoC) implementation which was developed as part of the thesis. It applies all suggested techniques on a hardware platform which was built using only minimal and cheap components to closely resemble the intended target devices. The goal was to have a working node in an HBA network which is able to carry out control tasks. A freely available virtual machine implementation for microcontrollers was used as the basis for the PoC and several extensions and modifications have been applied. The PoC was utilized in a small test environment

consisting of a small EIB/KNX network with a number of sensors and actuators. Access to the network was established by equipping the hardware platform with the basic circuit of the Freebus project [1]. A programming model and a complete tool-chain are provided and sufficiently documented.

The presented PoC delivers some interesting and promising results. Although some limitations according to performance and stability are present, it was shown to offer decent flexibility with a comfortable way of programming it. Several tests and experiments were performed which have shown its significant potential. The PoC can also can be seen as an excellent base for future improvements and studies.

The outline of the thesis is as follows:

In Section 2, some theoretical background about security is given. A distinction between vulnerabilities, attacks and threats is made. Security mechanisms are presented as a way to enforce a security policy. As a small digression, some common security mechanisms are described. Furthermore, specific security problems with low-end embedded systems are identified and attacks against such systems are classified. Problems related to the implementation of traditional security mechanisms are stated.

Section 3 provides an overview of Home and Building Automation (HBA) systems. Their application fields are outlined and typical demands on them are listed. Typical characteristics with respect to the organization of such systems are described. The functionality is divided into three hierarchically ordered levels, the used device classes are classified accordingly. SACs are identified as low-end ES which are used extensively in HBA systems. The section is wound up with some remarks on popular open HBA standards with a detailed description of LonWorks, BACnet and EIB/KNX.

An overview of currently available approaches to improve the security of a system is given in Section 4. Several static and dynamic software techniques are introduced, in some cases existing tools and methods are described. The techniques are evaluated in terms of applicability to low-end ES software. In addition, some hardware supported techniques are described.

Section 5 describes the proposed architecture aimed at providing application security in low-end ES. The concept is explained with the used techniques and their combination. Following, some suggestions about the possible implementation of the architecture are given. The intended advantages are listed at the end of the section.

The implemented Proof-of-Concept is outlined in Section 6. After explaining the big picture, a detailed description of the used hardware platform and software is presented. A reference of the programming framework along with several example programs is given. The used tools are explained with instructions about their usage.

Finally, Section 7 contains an evaluation of the implemented PoC, with spe-

cial considerations of its usefulness in common building automation tasks. The memory consumption is evaluated and measurements related to the performance of the PoC are presented. Observed stability issues are analyzed. The Freebus basic circuit is evaluated in its utility in an environment of retail devices, with focus on the produced signal levels.

2 Security

Generally speaking, security refers to the protection of data, services and hardware against unauthorized access and disturbance. As stated by Tanenbaum et al [2], security refers to properties of computer systems which are intended to provide dependability, confidentiality and integrity which can be described as follows:

Dependability is a measure for the trust we can put in a system to deliver its services. It can be further divided into the properties availablity, reliablity, safety and maintainablity.

Confidentially is present if information is disclosed only to authorized parties. That means, measures to prevent unauthorized parties from reading must be taken, such as the use of encryption for message exchange.

Integrity refers to the certainty that alterations to a system's data can only be made in an authorized way. This requires that unauthorized changes are at least detectable and, in the optimal case, recoverable.

Usually, when speaking about system security, the terms vulnerability, threat and attack are used to describe the relation between a system's security and a malicious party seeking to compromise it. The terms threat and attack are often used like synonyms although in the understanding of the author there is a significant difference. The nomenclature used in this thesis is strongly related to the definition used by Pfleeger in [3]:

A *vulnerability* in the context of computer security is used to describe a weakness in the implementation or the conceptual design of a system which could be exploited by an attacker. The attempt to exploit a vulnerability, including all necessary steps and prerequisites, is called *attack*. The risk that an attack of given type is launched against a system is usually referred to as *threat*.

2.1 Vulnerabilities

There is a vast number of possible vulnerabilities a system can contain. However, most of them can be assigned to one of only a few common vulnerability types. In the following, a selection of them is explained and examples of their possible exploitation are given.

A *buffer overflow* occurs when a buffer in a program is filled beyond its limits. This may result in memory regions being overwritten, which may be used by a buffer overflow attack. For example, if it is possible to overwrite the return address of a function, an attacker may inject arbitrary code as part of the buffer and, by manipulating the return address, make the program jump to it. Buffer

overflows mainly occur in low level programming languages like C/C++ which have no integrated boundary checking.

When a program, under certain conditions, unintentionally modifies memory locations during its execution, it is said to be vulnerable to *memory corruption*. Such actions are usually caused by errors in the software. Corrupted memory can cause a program crash or strange behaviour, depending on the value of the data which gets overwritten. Attacks may target at exploiting memory corruption vulnerabilities, for example to compromise the availability of a system by crashing a service it provides.

Race conditions refer to situations in the execution of a program where the outcome of an operation depends on the timing of other operations. Race conditions are often caused by deficiencies in synchronization mechanisms. They are often hard to trace down because they may occur in quite complex constellations.

A *format string vulnerability* usually refers to the careless use of a function of the `*printf()` family which is part of the C standard library. These functions take a number of variables together with a format string, describing how the variables should be formatted into a string. The resulting formatted string is returned. The problem is that there is a special formatting operator (`%n`) which allows the number of printed characters to be written to one of the variables. For instance, `printf("xxx%n", &n);` would write the value 3 to the variable `n`. If the format string can be influenced by an attacker, this operator can be abused to write arbitrary values to arbitrary memory locations.

Denial of Service (DoS) vulnerabilities can, when exploited, disturb or even stop a system in delivering a service. A common example is overloading a webserver by continuously sending a large number of fake requests, which makes the server unresponsive to other real requests. DoS attacks sometimes benefit from poor testing of systems under extreme workload conditions which may allow them to crash the system.

Figure 1 shows a breakdown of the distribution of vulnerabilities. The breakdown was determined by searching the US-CERT Vulnerability Notes Database [4]. Searches were performed over all entries of the year 2007. It should be noted that since the search engine only allows full text searches, double recordings may be included (and are in some cases reasonable because some reports describe a combination of vulnerabilities). Nevertheless, the numbers should give an adequate overview of the commonness of vulnerability types.

Apparently, buffer overflows have the biggest share of reported vulnerabilities. This has been the case for several years now, for example consider the breakdown presented in [5]. Despite a lot of efforts to eliminate them (like the use of OS-sided protection mechanisms or boundary checking in modern languages like Java and .NET) buffer overflows are still the biggest source of vulnerabilities.

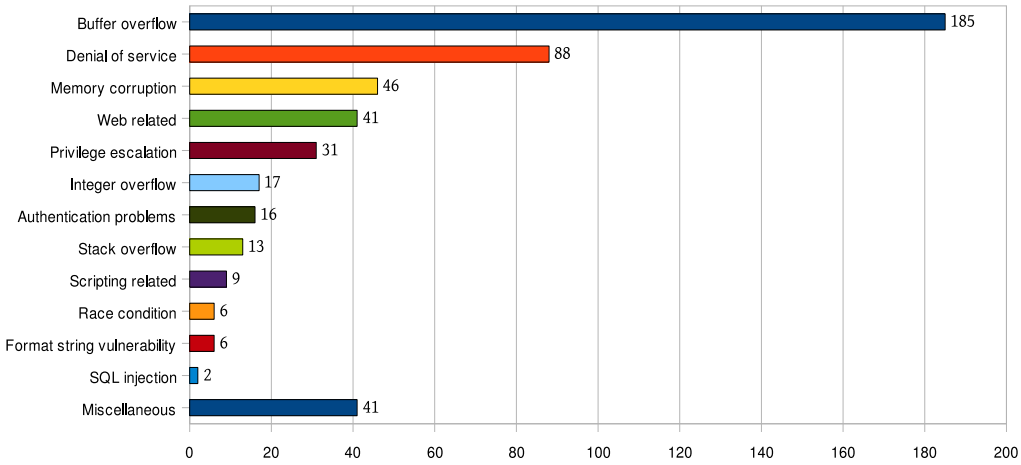


Figure 1: Number of vulnerabilities reported to the US-CERT Vulnerability Notes Database in the year 2007 for several common vulnerability types.

2.2 Threats

It has to be assumed that a system contains a number of vulnerabilities which can be exploited. From a security point of view, a system is exposed to several threats to a varying extent, depending on the nature and application of a system. Generally speaking, there are four types of threats [3]:

- Interception
- Interruption
- Modification
- Fabrication

Interception occurs when an unauthorized party gains access to a service or data. It also refers to the theft of information, for example a user's private data. The classical example of a related attack is an adversary eavesdropping the communication between two parties. Other examples would be the use of a network sniffer in a company's network or the use of a so-called keylogger to record all pressed keys on a victim's computer.

The goal of *interruption* is to corrupt or destroy data or to compromise the availability of services. Examples would be the deletion of data files or overloading of a server to make it unavailable for requests.

Modification refers to unauthorized changes of data or services. There are a lot of possible uses, for instance obfuscation of information or the activation of faked credentials. Examples for typical related attacks include SQL injections which may alter the content in a database.

Finally, *fabrication* describes the generation of additional data by a malicious party, data which would normally not exist. It includes the creation of false information as well as the generation of counterfeit communication messages. An advanced related attack would be a so called man-in-the-middle attack, where previously recorded messages are played back to a victim to cause certain reactions.

2.3 Security Mechanisms

As a requirement for building a secure system by minimizing the risk caused by threats, a *security policy* is needed. It precisely describes which rights the actors in a systems are granted and which actions are allowed and which are prohibited. Once a complete security policy is available, it needs to be enforced by applying *security mechanisms*. In the following, four important general security mechanisms will be described [2]:

- Encryption
- Authentication
- Authorization
- Auditing

Encryption makes information unreadable to persons who do not share a secret key or mechanism. It is used for example to transfer secret messages or to store confidential resources in a secure way. Encryption mechanisms can also be used to check for changes in data.

Authentication refers to the process of verifying the identity of a claimed identity, for example of a user, a client or a server. A common way to authenticate users is the use of passwords. Other methods include cryptographic signatures, biometric checks or automated challenges.

Authorization is used after an identity has been authenticated. It is used to determine the rights the identity is granted, for example to check if it is allowed to access certain files or to execute operations.

Auditing is used to log the actions of users on a system. This includes login times, file accesses, operations and so on. While auditing does nothing to improve security directly, it can be extremely useful for analyzing security breaches.

2.3.1 Implementation of security mechanisms

In [6], a distinction is made between functional security mechanisms and the security of their implementations. It is obvious that even the best security mechanism cannot provide good security if it has a poor implementation. It requires a flawless theoretical background as well as a proper implementation to be effective. But even if the mechanism is proven to be secure and well implemented, it is unrealistic to assume that attackers will only attempt to break it directly. It is likely that they will also look for ways around the used mechanisms, which may be found as weaknesses in the implementation and deployment and can be used to bypass or at least weaken the theoretical strength of security solutions.

Especially if a system's security is considered as a whole – taking into account its software and hardware – guaranteeing that the implementation of a system is secure is very difficult. The more complex a system is, the more unexpected side-effects can arise and unintended sources of information, which can be analyzed by attackers, may be present. Therefore, the implementation of security mechanisms can never guarantee full system security against all kinds of threats, partly because not all possible threats are known.

2.3.2 Common security mechanisms

To provide a small survey of important security mechanisms, some common techniques and implementations are explained in this section. Their basic principles are described along with some remarks on their practical use, origin and limitations.

Cryptography

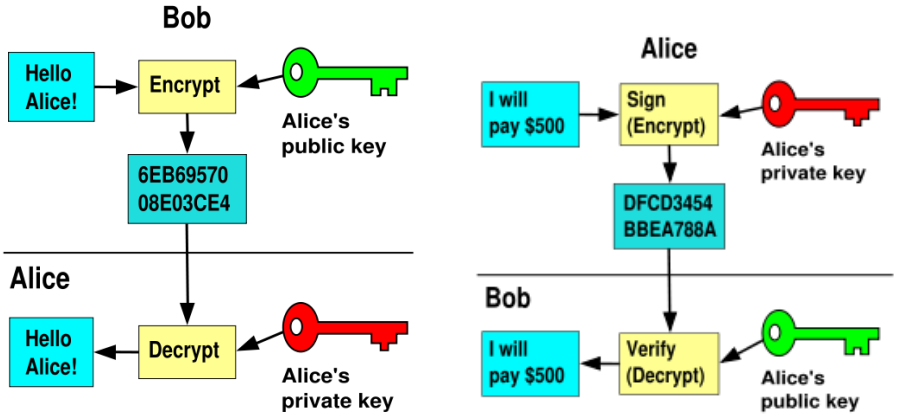
Cryptography generally refers to the practice of hiding information by translating it into a form only readable for designated parties. Modern cryptographic systems can generally be divided into two classes. If the key which is used for encrypting a message can also be used for decrypting it, the mechanism is *symmetric*. Otherwise, if the keys for encryption and decryption are different, it is *asymmetric*. Both classes have their advantages and applications [2].

In symmetric cryptosystems, the sender and the receiver share the same key to be able to exchange encrypted messages. The key has to be kept secret, which can be a significant problem because it has to be distributed among all participating parties before the actual secret communication can take place.

Popular examples of symmetric cryptosystems are the Data Encrpytion Standard (DES) [2] and its successor the Advanced Encryption Standard (AES), both standardized by the US government. Both cryptosystems use block ciphers, which

means they separate the clear text message into several blocks of the same size and apply the cryptographic operations to them. AES, also known as “Rijndael” (a portmanteau of the names of the inventors, Joan Daemen and Vincent Rijmen), is a very popular algorithm that is used worldwide. Its implementations are fast and require little memory.

In asymmetric cryptosystems, instead of using a single key, a pair of keys is used. The keys have to be related to each other, making sure that if one of the keys is used for encrypting a piece of information, it can only be decrypted with its corresponding key. Moreover, it is important that one key does not give any hint about the other. If this is the case, it is secure to make the key used for encryption public, for example by putting it on a website. This way, everyone can encrypt a message with this public key, but only the person owning the corresponding private key can decrypt it. This concept is called public-key cryptography, it is illustrated in Figure 2(a). The biggest benefit of public-key cryptography is that it solves the problem of distributing secret keys. The sender and the receiver do not have to share any secret information. Given that it is irrelevant which of the both keys is used for encryption or for decryption, public-key cryptography can also be used for digital signing. The sender can e.g. attach his name, encrypted with his private key, to an e-mail. The attachment can then only be decrypted with the senders public key, which allows the receiver to check the identity of the sender. The concept of digital signing is illustrated in Figure 2(b).



(a) Encryption (b) Signing

Figure 2: Illustration of encryption and digital signing using public key cryptography.

The concept of public-key cryptography was first introduced in 1976 by Whitfield Diffie and Martin Hellman. They published a paper explaining the Diffie-

Hellman key exchange protocol [7] . In 1977 Ron Rivest, Adi Shamir and Leonard Adleman publicly described RSA which was the first algorithm known to be suitable for signing as well as encryption [8] . RSA is based on the fact that there is no known method to efficiently find the prime factors of large numbers. The algorithm is still widely used today and is believed to be secure given sufficiently long keys and the use of up-to-date implementations.

Although public-key cryptography has several advantages over traditional symmetric cryptography, its implementations are significantly slower and they require more memory. For this reason it cannot be seen as a replacement for symmetric cryptosystems. But the combination of the two is very powerful and has a wide range of applications. One possibility is the use of public-key cryptography for exchanging a secret key which is used in a symmetric algorithm to encrypt messages. This concept was for example realized in the popular "Pretty Good Privacy" (PGP) [9], a program originally created by Philip Zimmermann in 1991 which can be used for signing, encrypting and decrypting e-mails.

Hashing

While cryptographic techniques are usually used to keep information secret between authorized parties, and therefore support the confidentiality of information, hashing is intended to support the integrity. Instead of making the information unreadable it rather aims at making even the slightest changes detectable. Hashing functions typically process a big amount of information to create a small hash value out of it, also simply called hash or checksum. This hash should, in the optimal case, be unique for every combination of input data. This is of course not possible in practice, because the generated hash should also have a practical length. But still, a quality hashing function should be designed to avoid collisions. Another important measure of the quality of a hashing function is its resistance to the computation of inverses. While the generation of the checksum has to be of low computational complexity to allow efficient implementations, the inverse operation, i.e. finding data which results in specific hashes, should be as hard as possible. Otherwise modified data could be prepared to result in the same hash value as the original. In fact, most hash functions are declared insecure as soon as the first collision attack is successful.

With a quality hashing function combining the properties described above, it is possible to provide information with guaranteed integrity. For example, it is a common practice on file servers to publish files along with their checksums as separate files, which allows the users to check the files for downloading errors. If hashing is used in combination with public key cryptography, information can be signed, meaning that the source and the exact content can be verified by a consumer.

Probably the most popular hashing function is the Message-Digest Algorithm 5 (MD5) [10] developed by Ronald L. Rivest in 1991 as a successor to MD4. It can be used on data with arbitrary length and generates hash values with 128 bits in length. Although it was shown to be vulnerable to collision attacks by chinese scientists in 2004, it is still widely used. Examples of more advanced and more secure hash functions are the secure hash algorithm (SHA) in its more recent versions (SHA-512 or better) [11] or RACE Integrity Primitives Evaluation Message Digest (RIPEMD) [12] .

Authentication

To verify the identity of a subject, for instance a user, process or machine, authentication mechanisms are used. In order to enable a subject to be authenticated, it has to provide some kind of information related to its identity. Generally the information can be something the subject knows, like a password, something the subject has, for example a keycard or something the subject is, like biometrics (e.g. an iris scan). Each of these options has its benefits and drawbacks, that is why especially in security critical applications, a combination of authentication mechanisms is used.

2.4 Evaluating Security

Since computer security is an ability composed of several properties, it is difficult to measure the grade of security or in other words the resistance against attacks to a system. Unlike, for example, safety related properties like availability, security related properties can generally not be expressed in a numerical form. Nevertheless, it is often necessary to have some kind of quantitative measure about the security of a system.

Considering only the theoretical aspect of a security mechanism, there is often a mathematical representation available. In this case, mathematical reasoning methods can be used to prove certain properties or give an indication about the complexity of breaking the mechanism. But as mentioned earlier, it is naive to make any assumptions about the security of a computer system while only looking at its theoretical basis.

A general approach to security evaluation is represented by existing security standards such as the European ITSEC, the American TCSEC or the combined international *Common Criteria standard* [13]. The Common Criteria standard was developed to provide a common base for security evaluations, partly to avoid the need for multiple certification processes of systems.

It defines a set of constructs which classify security requirements into related sets called *components*. Each component identifies and defines any permitted

operations, the circumstances under which it may be applied and the results of the application. The components are grouped into *families* of components which share security objectives. Families are further grouped into *classes* which are families which share a common intent. Eleven functionality classes are defined, among them are "Cryptographic support", "Identification and Authentication" and "Privacy".

For a specific target of evaluation, a set of components, describing the IT security objectives and requirements which should be met, may be selected. These collected requirements are called the *Security Target (ST)*. In the evaluation process of a system, the requirements defined in its ST are checked using one of seven Evaluation Assurance Levels (EAL1-7). EALs provide a increasing scale which balances the level of assurance obtained with the related cost and feasibility. The developer of a system is free to choose the desired EAL.

The evaluation can be performed in licensed laboratories which exist in many countries spread all over the world.

2.5 Security in low-end embedded systems

Low-end embedded systems are typically found in areas where very strict demands in terms of cost, power consumption and resources have to be met. These requirements are also often tightened due to the large needed number of such systems. They are characterized by microcontrollers with low computational power, few resources and a small feature set, just powerful enough to fulfill their destined task. More specifically, in this work, systems using 8- or 16-bit microcontrollers clocked at less than 25 MHz, and with less than 100 kilobytes of program memory, are considered being low-end embedded systems and represent the target of the described investigations.

2.5.1 A Scenario

Due to the widespread deployment of embedded systems as well as the importance of the tasks they take over, security flaws in such systems may have severe consequences which are not obvious in the first place. For example, consider a scenario depicted by Koopman [14], where embedded systems are used for household thermostats which are connected to the Internet. The idea is interesting, as it allows the inhabitants of a house to regulate the temperature of their flats or houses even when they are not at home. For example, it could be used to turn off the heating during a working day and turn it back on again an hour before coming home, resulting in an already comfortable temperature in the residence at arrival.

If an attacker is able to gain control over a household thermostat, he could already cause serious trouble. He could waste energy by turning on the heating or

air conditioning when unnecessary, resulting in increased cost. But besides that, he could disturb the comfort of the inhabitants or even do damage to the residence, for example by completely turning off the heating in cold winters, causing pipes to freeze.

Controlling household thermostats could also cause damage on a bigger scale. If an attacker controls a lot of them, he could take influence in the total power consumption of a region. For example, turning on a lot of air conditioning systems at the same time can seriously increase the demand on electricity and could even cause power-grid failures. This example shows how serious security flaws in embedded systems can be, even though the risks are not obvious in the first place.

2.5.2 Attack Classifications

In general, the same general threats as previously described also affect low-end embedded systems. Additionally, they must often operate in untrusted environments where physical access by attackers is given. But due to the usually very limited resources in such systems, traditional protection mechanisms are often infeasible. Special requirements impose extra possibilities for attacks, like the ability to download and execute code which is often required for maintenance. Finally, embedded systems are often designed in complex processes which includes the consideration of many aspects of which security is not the most important at design time. Therefore, some special security threats must be considered ([6], [15]).

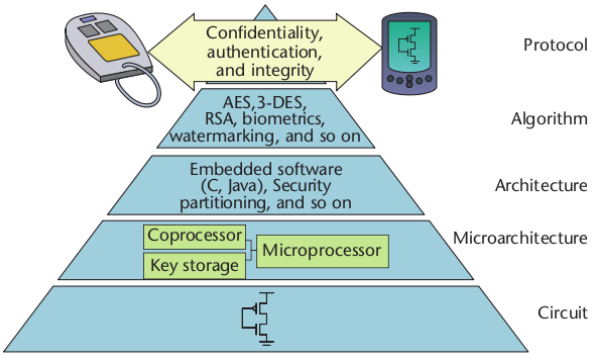


Figure 3: The embedded systems security pyramid as introduced in [15].

It is important to note that an attacker cannot be expected to only try to break the cryptographic mechanism, especially with embedded systems. It is often much easier to exploit implementation issues that allow bypassing or weakening the mechanisms. Therefore the security of a system may never be considered solely

by its theoretical capabilities but also as a whole including its software implementation and physical realization. Hwang et al [15] defined a security pyramid as depicted in Figure 3 which describes five levels of embedded systems security:

Protocol level: The protocols used in embedded devices.

Algorithm level: Cryptographic primitives and application specific algorithms.

Architecture level: Hardware/software partitioning and system organization.

Microarchitecture level: Hardware design of modules.

Circuit level: Transistor-level security and packaging.

They state that, in order to make an embedded system secure, the threats in all five levels must be considered. If only one level is vulnerable to attacks, even the strongest security mechanisms on other levels cannot be effective.

Ravi et al [6] classify attacks based on the means used to launch them. This classification is also adapted here. Three categories are used:

Software attacks interfere with a systems software implementation.

Physical or Invasive attacks use physical intrusion or manipulation at some level.

Side-channel attacks are based on observing properties which are measurable during operation.

In each category, attacks can be passive or active, the former only to collect behavioural or confidential information, the latter to also manipulate a system. In practice, attackers use a combination of various attack techniques. In the following characteristics of the described attack categories will be explained and some examples will be given. Figure 4 gives an overview of the attack types and their described characteristics.

Software attacks

Software attacks use regular communication channels of a system to exploit weaknesses in the architecture of a system. Such weaknesses are often caused by insecure programming, e.g. the use of problematic functions or the absence of parameter validation. There are countless ways of how vulnerabilities are exploited, but typically some form of code injection is involved. This way, an attacker can force a system to execute operations he dictates. The most popular code injection attack is the buffer-overflow. A vulnerability like this is particularly common in

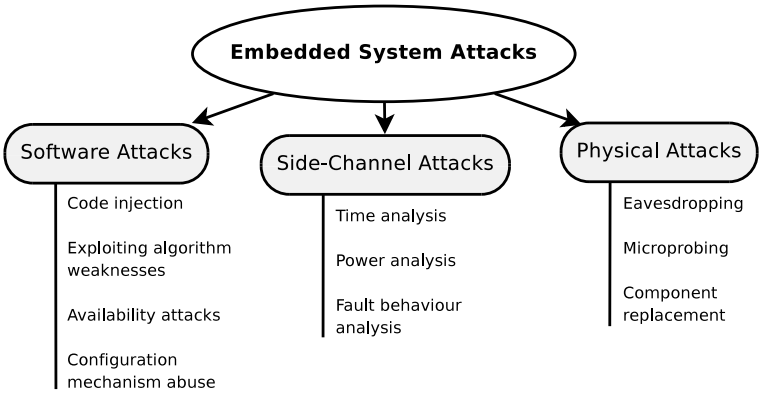


Figure 4: Attacks on embedded systems.

programs written in low-level languages like C, which is naturally a very common language in low-end embedded systems programming. Making sure that code is without such flaws is a difficult task which requires comprehensive programming experience. Some weaknesses can only be exploited through very complex interaction of several software components making the task even more difficult.

Besides implementational problems, also weaknesses in the underlying algorithms can be the basis for security attacks. This may especially be the case for adapted or self made cryptographic algorithms which are not proven to be secure. It is generally hard for a developer to imagine how somebody may try to outsmart his algorithms. Additionally, developers are often forced to develop their applications rapidly, which leaves no time for extensive security analyses.

Software attacks only targeting at compromising a system's operation, are referred to as availability attacks. Such attacks may even work in the absence of any security flaws in the software by overstressing the available resources. For example, a networked system could be made unavailable for processing any valid requests by flooding it with fake requests, a typical DoS attack. Additionally, if such a high frequency of requests is very improbable at normal operation, it may not be very well handled (and tested) in the application code and may cause a system to crash.

A special issue that is present in embedded systems is caused by the often needed functionality to download and execute application code during operation. This functionality may be abused to install malicious programs. This is especially dangerous if the attacker has physical access to the system, which is often the case.

Physical attacks

Physical attacks aim at collecting information or interfere with a system on a physical level, by using probes or by applying modifications to the hardware. Although often sophisticated and expensive equipment is required, they nevertheless represent a threat that cannot be neglected. Such attacks are to some extent a speciality of embedded systems. Normal workstations or servers can usually be protected against unauthorized physical access while embedded systems are very often required to work in an untrusted environment.

The most obvious way to collect information by physical probing is by eavesdropping inter-component communications. If, for example, an external memory chip or co-processor is connected to a processor, information can be collected by listening to the connecting media. This is especially easy when standardized protocols are used for the data transfer and the data itself is unencrypted.

Physical attacks at the chip level are harder to deploy since they require an expensive infrastructure and considerable knowledge. These attacks may require difficult activities like chip de-packaging, layout reconstruction or the use of e-beam microscopy. However, they can be used to collect important information that can be the basis for successful non-invasive attacks. Figure 5 shows the magnified surface of a chip which is prepared for microprobing.

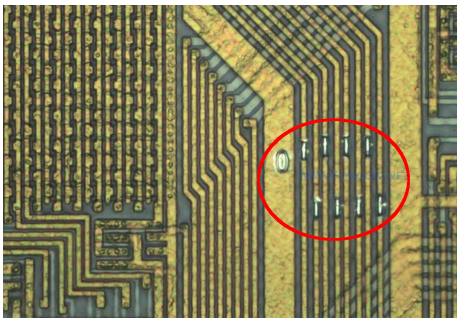


Figure 5: Microprobing example: Eight data-bus lines where exposed on a chip surface using a laser. (Source: http://www.flylogic.net/)

Instead of interfering with existing hardware components, they may also be replaced or extended to change the behaviour of a system. This way, for example, a ROM chip could be replaced to change certain parameters of a system. One popular example of such attacks are so-called mod chips which are used to disable copy protection mechanisms in video game consoles. These mod chips need to be attached to the consoles internals and are able to modify or disable security mechanisms.

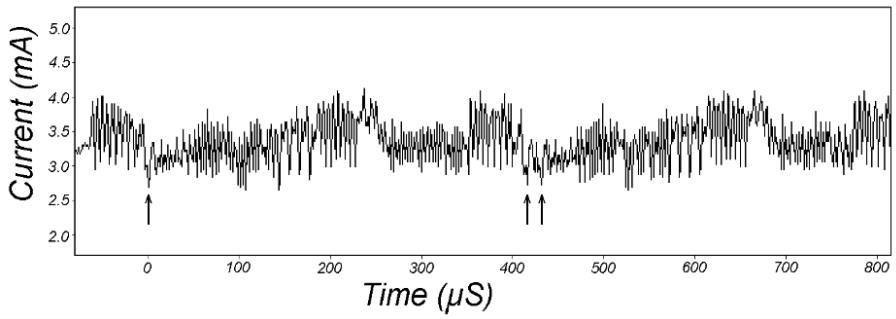


Figure 6: Power consumption measurement of a smart card performing a DES encryption operation [16].

Side-channel attacks

Side-channel attacks target at observing external parameters of a system to collect information about its internals. These parameters include execution time, power consumption or fault behaviour.

The basis for power analysis is that the power consumption of any hardware circuit is a function of the switching activity at the wires inside it. The switching activity is on the other hand data dependent, therefore it is possible to use statistics about the power consumption of a system to determine keys used in cryptographic algorithms. It has been shown that this analysis can be used to break embedded systems such as smart cards in an efficient way. Figure 6 depicts an example trace of a smart card performing a DES operation. The arrows mark visible details of the operation, according to [16], DES key register rotations.

Listing 1: Simple (problematic) authentication example (pseudocode).

```
boolean Authenticate(username, password)
{
    if user_exists(username)
        return check_password(username, password);
    else
        return false;
}
```

Another way to collect information about a system through its behaviour is the use of timing analysis. For example consider a simple authentication routine as depicted in Listing 1. In the example routine, if a given user exists, the provided password will be checked, if not, the routine immediately returns. It can be assumed that the function `check_password(...)` takes more time than just returning false. This way it is possible to scan for valid user names just by analysing the time it takes the system to reply. More sophisticated analyses com-

bined with statistics can reveal the number of multiplications and divisions in a cryptographic algorithm, since these operations take a variable number of cycles based on the data inputs on many processors.

2.5.3 Problems with traditional security mechanisms

As stated before, there are several sources of problems when trying to implement traditional security mechanisms on low-end embedded systems. Ravi et al [17] identified a number of problems which may limit the applicability of such mechanisms:

Processing Gap: Security mechanisms are often complex and therefore computationally intensive or require the processing of high data rates within a reasonable amount of time. Usually, embedded systems, especially low-end ES which are focused in this work, are not powerful enough to meet these demands.

Battery Gap: Some embedded systems are battery-driven which requires the software to be efficient in terms of power consumption. Since power consumption is strongly related to the processing complexity, the use of security mechanisms can increase the power consumption of a system significantly.

Tamper resistance: As described in Section 2.5.2, embedded systems face threats related to the physical implementation of a system. It is therefore required – in addition to a proper software implementation – to also consider the vulnerabilities exposed by the hardware implementation. It may be necessary to use tamper resistance mechanisms to protect the software implementation.

Flexibility: Security mechanisms are often evolving, since new threat scenarios arise over time. A system should be flexible enough to be adapted to new and extended techniques. On the other hand, increased flexibility may make the assurance of a system's security unnecessarily difficult.

Assurance Gap: Secure systems are required to work reliably under the presence of dedicated attacks which seek to exploit vulnerabilities contained in the system. It is therefore significantly more complex to design a secure system than one which just works as it is required to. The task is also complicated due to the increasing complexity of embedded systems.

Cost: As stated before, security mechanisms are often of high complexity. Therefore, the design and implementation are also time-consuming and cost intensive tasks. Especially if implemented on an embedded system, where

many surrounding factors have to be considered in judging a systems security, cost may also be increased due to the requirement of special protection and testing.

2.6 Securing the Host and the Code

With the evolving concept of mobile code, which is code that gets migrated between and executed on different hosts, the security of both the executing host as well as the code together with its data has to be considered (Figure 7). On the one hand, the code may contain confidential data which the host should not be able to read. On the other hand, the code should not be able to carry out any undesired actions on the host or collect secret information stored on it [2].

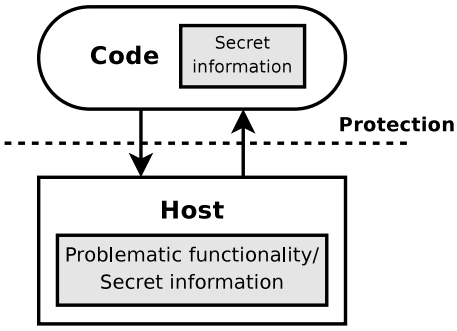


Figure 7: With mobile code, the code as well as the host executing it have to be protected.

The protection of the code is especially difficult because it can never be guaranteed that the host behaves as it is expected. Code signing and encryption mechanisms cannot be used effectively since the host naturally has full access to every piece of the code.

Protecting the host is the more common problem one is faced when dealing with mobile code. The host is exposed to several security risks. For one thing, secret information like cryptographic keys may be stored in some memory regions on the host. If code being executed on the host is able to access the same address space without any restrictions, it can easily acquire this kind of information. If it is also able to access functions for output or some kind of communication, this information can be gained by an attacker. Besides the risk of stolen information, the downloaded code may also disturb the operation of the host, or in case the host is part of a communication network, a network of several nodes.

According to [18], there are three practical techniques for securing mobile code execution. The first and nowadays very popular method is to restrict the

downloaded code in its actions. This is done by limiting its permissions and providing only a small set of functions the code can issue. This approach is typically referred to as the *sandbox model*. *Code signing* is a technique which extends code with information about its source and its trustworthiness. With this information, the executing host can decide if the downloaded code should be executed or not. The third approach, *firewalling*, uses specific rules which refer to executable properties to determine if and how downloaded code is executed as they enter a trusted domain.

3 Home and Building Automation systems

Home and Building automation (HBA) systems are generally concerned with improving the interaction between in-house devices and carrying out control tasks [19]. The devices taking part in these systems are connected via a shared medium, often a physical bus line or radio in modern systems.

In this thesis, HBA systems are presented as a case study of a growing field of technology which is in its current state does not consider security aspects to a large extent. Furthermore, a device class which consists of low-level embedded systems is identified and used as a reasonable application area of the presented investigations and developments.

Automation of large scale functional buildings (like company buildings with extended functionality and possibly many occupants) is of particular interest for the application of HBA technology because it offers big potential for cost and energy savings. Although the construction costs of such installations compared to standard electrical installations are significantly higher, extensive savings can be identified when considering the whole building lifespan. According to [19], the operational cost of a building over its lifetime is about seven times the initial cost for the construction. Therefore, even small savings can justify the increased financial investments and effort which is required for the installation of such systems, especially when considering the increased comfort and flexibility that is provided.

The following list describes some common domains to be replaced by or extended with HBA technology.

- Lighting and shading
- Heating, ventilation and air conditioning (HVAC)
- Household devices (“White goods”)
- Home-Entertainment devices (“Brown goods”)
- Communication systems
- Security and access control
- Safety alarm systems

A distinction between homes and buildings is made. Homes typically refer to personal houses and flats, with limited size and with only a few inhabitants. On the other hand, buildings in this context refer to large scale functional buildings, usually company buildings with extended functionality and possibly many occupants, like day time workers. While HVAC, lighting and shading are the “traditional”

applications for building automation systems, the integration of household and home-entertainment devices is becoming more attractive for home automation.

Music and light that follows a user while moving through a residence is a good example for an interesting scenario that could be made possible with HBA technology. Remote administration of building functions using, for example, the Internet is another tempting application. Additionally, HBA systems may also be used to increase the safety and security by integrating alarm systems and access control. Besides the mentioned improvements in efficiency, comfort and cost, it should not be forgotten to also look at the potential such systems have as a supporting technology. They can be used to enable or support an independent life for handicapped or elderly people.

3.1 Demands on HBA systems

A number of general demands on HBA systems can be identified. These demands are described in this section with some comments about how to meet them.

Dependability: Since HBA systems are often intended as a replacement for traditional electrical installation equipment, it is important to provide comparable dependability and lifetime. Since relatively complex systems are used (for example, consider a simple light switch compared to its microcontroller-driven, bus compatible counterpart), extensive testing of the components and quality assurance is needed. A system should also be designed in a way to provide graceful degradation. This means that upon failure of parts of the system it continues to operate, providing a reduced level of service rather than failing completely.

Flexibility: As stated before, the integration of HBA systems is particularly interesting in big office buildings which have a high power and resource saving potential. Depending on the size of the building, such systems can grow very complex. Besides, the demands on the functionality are likely to change over some years. Thus, the systems need to scale and should be easy to extend. These demands are usually met by dividing an automation network into several subsections, utilizing the implied spatial locality of such systems. For instance, sensors and actuators which are situated on the same floor of a building are in most cases related and require only little communication to systems on other floors or to centralized structures. Another important demand is the possibility to connect a companies IT-infrastructure (e.g. Intranet) to the automation system.

Response time: HBA systems are intended for the transmission of control signals. For this reason, their focus is set on responsibility and reliability.

Therefore, the data rates being used for communication are usually relatively slow which makes such systems unsuitable for the transmission of large amounts of data (which is not their purpose anyway). Nevertheless, a reasonable response time, even in situations of high activity, is often desirable and in case of critical applications, a must. For example, while it may acceptable for a centralized building control center to receive temperature values of rooms within minutes, a light should be turned on within a second at the latest when a light switch is pressed. Therefore, an HBA system needs to be able to function even in situations with high traffic by prioritizing different types of signals.

Interoperability: Different building automation technologies may be used for different tasks and purposes. For example, for fire alarms and other critical event reporting, it is often necessary to operate a separate network for safety reasons. To this end, an HBA system should offer interoperability with other common standards. There are several ways to achieve this, the most desirable one being the adherence to open standards.

Usability: Since it cannot be expected that the users of HBA systems receive extensive training on the operation of HBA devices, their usage should be as simple and self-explanatory as possible. This is especially the case for private tenements.

3.2 Typical system characteristics

Typically, HBA systems consist of a collection of embedded devices along with some control nodes, couplers and gateways. They are connected via a communication medium, in many cases this medium would be a common physical bus line. Depending on the size of the system, it is often also separated into several interconnected control networks. Especially in new buildings, installation wiring using twisted pair cables is usually the cheapest and most stable interconnection solution. It also has the benefit that bus lines can transport the required energy for the connected devices along with the communication signals. Since it is sometimes necessary to upgrade existing buildings and also due to their ease of installation, wireless solutions become more popular. Other possibilities include the use of power lines.

The nodes in such networks communicate via a common protocol. Usually a Carrier Sense, Multiple Access (CSMA) mechanism is applied to deal with collisions. As stated before, in order to maintain a system's scalability, a network may be subdivided. For this purpose, subnetworks need to be interconnected by special coupler nodes carrying out filtering tasks. In heterogeneous networks, gateways

can be used for protocol and name space translation. If control networks are spatially separated, high-performance network backbones may be used for their interconnection.

Nodes in the control networks store and exchange process data values. These values can represent boolean values (e.g. switch states), numerical values (e.g. temperature, luminance) or other data types like floating point numbers or strings for various purposes. As a layer of abstraction, usually the concept of a *data point* is used. A data point is a logical representation of a value contained in the network and is associated with a unique identifier and data type. It may refer to an actual physical setting or an abstract value.

Each node can be associated with one or more data points. These assignments and the interactions between the data points are usually configured at installation time. Simply put, this includes settings like which switch should control which light. To allow such settings, there must be a way to either directly address a node with a configuration tool or the node implements some sort of configuration mechanism itself. A configuration tool can also help managing complex systems.

3.3 System structure

The system functionality of an HBA system can be divided into three hierarchically ordered levels [19]:

1. the field level,
2. the automation level and
3. the management level.

At the field level, interaction with the physical environment is done. Sensors are used to collect information, like measurement data, and actuators are used to take influence, e.g. controlling devices. The automation level is used to control the sensors and actuators and to realize interrelations. Global configuration and management tasks are carried out in the management level.

According to [20], in modern HBA, a two level model is more appropriate to realize such systems, where the tasks of the automation level are split and together with the field level form the *control level* and with the management level the *backbone level*. This division as depicted in Figure 8 is reasonable since field devices become more powerful and can take over more advanced functionality. On the other hand, cheap IT hardware can be used to take over management tasks, making specialized automation hardware obsolete in many cases.

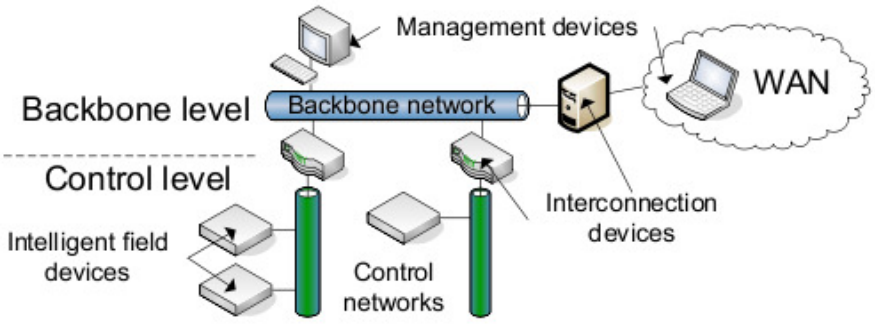


Figure 8: A two level architecture for HBA [20].

3.4 HBA device classes

Likewise, the devices used in HBA systems can be classified. In the field level *Sensors, Actuators and Controllers (SACs)* perform measurements of environmental data, affect physical parameters and hold controller functionality. Examples for such devices are switches, light sensors, temperature sensors, door openers or controllers for window shutters. SACs are usually characterized by cheap low-profile components with minimized functionality and degrees of freedom.

In the backbone level *Interconnection Devices (ICDs)* link different networks or different segments of a network together and *Configuration and Management devices (CMDs)* are used to maintain and configure the system. Tools used for the diagnosis of errors and misconfiguration in the network can also be classified as CMDs. Besides specialized (and therefore often expensive) components like gateways and bus couplers, standard IT technology (e.g. IP infrastructure) is often adapted in modern networks in the backbone level. The main benefit of IT technology is that it is cheap and well-engineered and can satisfy high performance demands.

SAC devices play an important role in HBA systems and are usually very large in number. This is one reason why, in order to be profitable, strict requirements are applied to them. They have to be cheap, both in the production as well as in operation, asking for low power consumption and long lifetime. These requirements hamper the adoption of computational powerful and resource rich processors, due to their usually implied complexity and cost. Therefore, SACs are in most cases devices with very limited processing power and memory, featuring only necessary functionality.

In [20], the requirements on SACs are summarized. A low cost 8 or 16 bit microcontroller unit equipped with a few kilobytes of memory is stated as being

sufficient for such a device. Additionally, a point to point interface for configuration and application uploading is required along with interfaces for communication. Also, low power consumption is of major concern as well as small size and robustness.

3.5 Popular open HBA standards

Several open home and building automation standards are available for varying purposes. Some have been developed for specific application areas while others are intended to provide complete solutions.

A popular example for a specific standard is the Digital Addressable Lighting Interface (DALI) [21] published as an IEC standard, which is widely accepted for lighting applications. It can be used to control up to 64 dimmable flourescent lights in a loop. Another example is M-Bus [22], a European standard which is designed for remote reading of gas or electricity meters. Since such solutions often carry out only specific tasks and do not necessarily require a high degree of flexibility they may be designed to be very efficient. This efficiency can generate several technical and economical advantages over general solutions, for example lower cost and energy consumption. For such specialized systems it is very important to be able to interoperate with other systems, possibly operating above them. Therefore, openness, in the sense that everybody can interact with a system without having to rely on the original manufacturer, is an important feature.

Among open standards which aim at providing complete solutions for HBA, LonWorks, BACnet and EIB with its successor KNX are the most popular ones [19]. While BACnet and LonWorks have considerable importance in the worldwide market, KNX is particularly successful in Europe. These standards have different origins and initial purposes which is also reflected in the application area they are focused on today. In principle, these standards aim at providing a complete solution on all HBA levels and could be designed entirely non-open. However, openness has some significant advantages for the customers, integrators as well as the manufacturers of such systems and is therefore an important factor for the commercial success of such systems. The customer is protected of the so-called vendor lock-in, meaning that he does not need to rely on a single manufacturer. Given that interoperability is assured, integrators are able to mix devices from different manufacturers for increased performance and cost efficiency. Another advantage is that small manufacturers have the possibility to concentrate on niche products.

The application areas of complete solution standards are widespread which makes them an interesting target for security improvements. In the following, an overview of LonWorks, BACnet and KNX/EIB is given.

3.5.1 LonWorks

LonWorks [23, 19] is a networking solution for automation and control applications developed by the Echelon corporation. It is suitable for building and operating powerful and widely ramified networks. The basic rules for communication in a LonWorks network are defined by the LonTalk protocol which has to be supported by any device used in such a network. It is implemented in the so called Neuron chips designed by Echelon as well as on devices by some other manufacturers (e.g. Loytec).

The LonTalk protocol was accepted as a standard for control networking (ANSI / CEA-709.1-B) in 1999. The standard was used in many subsequent standards in the fields of train service, semiconductor manufacturing and, in 2005, for building automation in the European building automation standard. It has also been adapted as transport protocol for the BACnet ASHREA/ANSI standard for building automation.

LonTalk was designed for applications involving sense, monitor, control and identification functions. Its key features are high reliability, variety of communication media and low response time. It was also created to allow low cost production of compatible devices. While the LonTalk protocol is an open standard, interoperability among LonWorks devices by different manufacturers is enabled by guidelines and profiles defined by the LonMark Interoperability Association. A certification program has been put in place by Echelon to assure compatibility between devices.

In a LonWorks network, a *domain* describes the entire routable address space. It consists of up to 255 *subnets* with a maximum of 127 nodes each, which makes a maximum number of 32.385 nodes in a domain. All nodes in a subnet must either be on the same communication channel or on channels connected by bridges. Subnets can be connected by routers, usually with a fast bus topology as a backbone.

Individual nodes in a LonWorks network are addressed by their unique 48-bit identifier, which is carried by all devices and assigned during manufacturing. This addressing mode is particularly important for management and configuration tasks. Regular unicast communication is handled through logical subnet and node addresses. Additionally, nodes can be part of multicast groups which may be used to group several nodes of similar or related functionality. A domain can host up to 256 multicast groups and each node can be a member of up to 15 groups.

Communication between the nodes can be either implicit or explicit. The preferred way of data exchange is the implicit way by using *network variables* (NV). NVs can be defined in the node application programs and logically interrelated across the whole network. Each node can hold up to 62 NVs with a maximum length of 31 bytes. They can be either input or output variables. Writing to output

variables in the application program triggers the distribution of the new value to all related input variables in the network automatically. There are different ways of how NVs are related to each other. In some systems this happens at manufacturing, or using a configuration system at installation time. In homogeneous networks it is possible to use automatic configuration techniques (ISI protocol).

In some cases the use of NVs is insufficient, for instance when data with a length of more than 31 bytes needs to be exchanged or when a response message is required. With explicit messaging, it is possible to send messages with a length of up to 229 bytes and an additional request/response mechanism is provided. The communication process is carried out in the application program.

For the exchange of NV values as well as explicit messaging, the protocol features several message services, to meet varying requirements:

Unacknowledged: In unacknowledged mode, messages are sent whenever a node determines it is appropriate. No responses from receiving nodes are perceived. This the most commonly used message service and requires the minimum network bandwidth.

Unacknowledged/Repeated: Identical to the unacknowledged mode, but each message is repeated a configurable number of times.

Acknowledged: Messages are sent and acknowledgements are collected from each receiving node, possibly resulting in a retransmission if not all acknowledgements are received. This service is used in applications where certainty about message reception is important. The available bandwidth is reduced. If acknowledged service is used with multicast groups, the group may have a maximum of 63 members.

Priority: Several priority slots can be allocated on a channel and assigned to nodes for transmission of critical messages. Only one node should be assigned to a slot. With this service, communication bandwidth is reduced because in the priority time slots block the transmission of spontaneous messages and it should therefore be used rarely.

Several physical media are generally supported for communication using the LonTalk protocol. They include:

- Twisted pair (TP)
- Power line (powered and unpowered)
- Radio frequency (RF)

- Coaxial cabling
- Fiber optics

In LonTalk these media are referred to as channels, they can be split up into several segments. The 78.1 kb/s free topology TP profile is the most popular channel. It enables physical segments of up to 500 meters using low cost TP cable and can also provide link power.

Since many protocol parameters are free to choose for the designer, channel profiles have been defined to describe the use of certain communication channels more specifically. This way, by dedicating to the profiles, interoperability on certain communication channels can be achieved by different manufacturers.

The Neuron chip is a powerful component which includes almost all essential parts of a network-enabled device. It is a system-on-a-chip composed of three interconnected processors with internal ROM, RAM and EEPROM. It features an 11 bit wide I/O connector along with a 5 bit wide communication interface for connection to a LonWorks network. Only one of the three processors actually executes the application program while the other two are used for handling the LonTalk protocol processing. Applications running on a Neuron chip only need to implement the application layer of the network stack, everything else is taken care of by the LonTalk implementation on the chip. A large family of Neuron Chips is available with differing speeds, memory type and capacity, and interfaces. Most LonWorks devices are equipped with a Neuron chip, which ensures compatibility.

To ensure interoperability of devices from differing manufacturers in LonTalk networks, a common application framework is defined, using the concept of Standard Network Variable Types (SNVTs). SNVTs are described as an extensive list of common physical measurement types. They can be accessed in a standard way in a LonTalk network. When two devices refer to the same SNVTs in their implementation, they can be bound together.

Additionally, standard configuration property types (SCPTs) define a common way of accessing configuration parameters and standard functional profile templates (SFPTs) are used to describe application specific interaction of NVs, configuration properties, defaults and power-up behaviours. All these definitions and guidelines are not part of a formal standard but are freely available.

3.5.2 BACnet

BACnet [24, 19] is a data communication protocol for building automation and control networks. It is equally applicable for management and automation tasks, it can for example be used for HVAC, lighting control, building security and fire detection systems. The protocol defines the application and network layer, the

underlying layers can be handled by several different technologies. BACnet is an ISO global standard, can be used by anyone and is free of any licence fees.

The development of BACnet started in 1987 by the American Society of Heating, Refrigerating and Air-Conditioning (ASHRAE). The standard was originally published in 1995 as ANSI/ASHRAE standard. After some updates in 2001, it was approved as ISO standard 16484-5 in 2003. It has thereafter also been accepted as a CEN standard.

BACnet was designed to be applicable to the general needs of building automation systems. One of its main goals was to establish a vendor-independent standard for data communication in systems for building automation and as a consequence eliminating the fear of being bound to a single hardware vendor. It was designed with special regard to the communication model without defining rules for the transportation of data and rather adapting existing physical, data link and networking standards.

BACnet systems are divided into *segments*. Segments can be coupled by repeaters and bridges and form *networks*. Routers can be used to connect several networks together to form an *internetwork*. There is only one path allowed between any two devices in an internetwork.

A BACnet system may also consist of several networks based on different underlying technologies. The routers used for interconnecting different networks need to be able to learn the topology they are situated in. Figure 9 shows an example of such a heterogenous configuration.

Each device is assigned an address which consists of two parts: the first two bytes specify the network number. Routers route packets based on the network numbers. The second part is the local address which is up to 255 bytes long. The format of the local address depends on the underlying link layer medium, it can for example be an IP address if BACnet/IP is used.

In BACnet, *objects* represent physical inputs, outputs and software processes. They are a collection of information related to a particular function and can be uniquely identified and accessed over a network. Objects hold all information available in a BACnet network, including measurements of physical conditions as well as abstract information like calculations. BACnet objects can be accessed in a transparent way over the network.

To describe an objects behaviour and to affect its operation, it provides a set of *properties*. Some properties are read only, others can be altered.

In BACnet, 23 standard object types are defined, specifying their behaviour and which properties they provide. The set of objects represents much of the functionality typically found in building automation and control systems. The only object a BACnet device is required to implement is the special *Device* object. Other objects may be freely chosen by the manufacturer to achieve the desired device functionality.

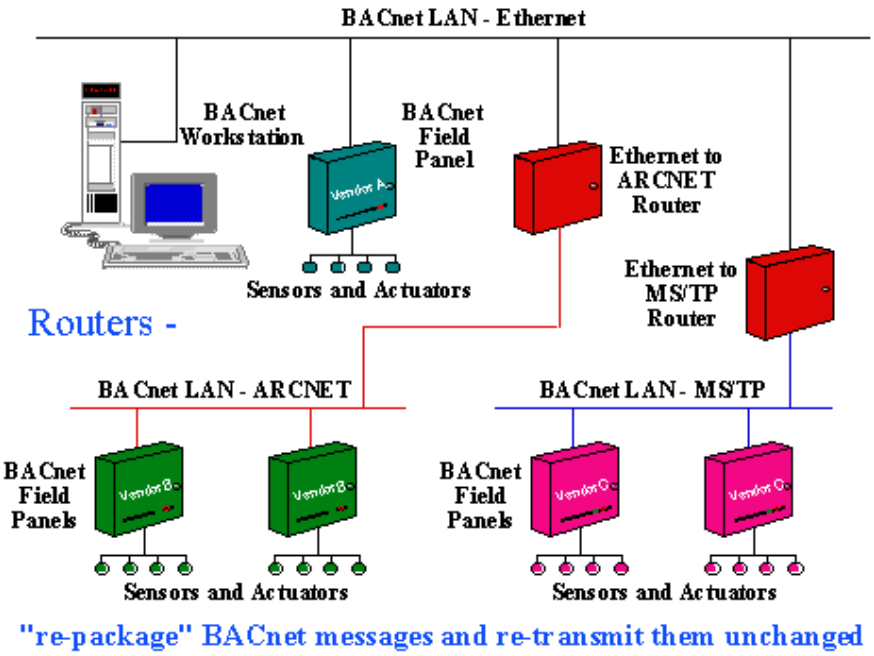


Figure 9: Example of a heterogenous BACnet network [24].

For accessing and manipulation objects, BACnet *services* specify messages for exchanging information and for more advanced functions. The most common services are `ReadProperty` and `WriteProperty` for accessing properties. In this context, devices hosting information which is used by other devices are referred to as servers, for example sensors offering a quantified physical value. The devices which request such information are called clients. Typical clients would be operator controls or management stations. A number of services for accessing properties of objects are specified, among them are services for reading and writing properties as well as for object management.

Although the BACnet standard is in principle independent of its message transportation, several network types have been standardized for the exchange of BACnet messages:

- Ethernet
- ARCNET
- Master-Slave/Token-Passing (MS/TP) protocol
- Point-to-Point (PTP) protocol

- LonTalk

The MS/TP and PTP were defined as part of the standard for low-cost and dial-up communications. BACnet/IP was standardized to transport BACnet messages over the Internet Protocol by extending the protocol stack with the "BACnet Virtual Link Layer" (BVLL) which allows the use of underlying protocols as virtual datalink layers. This virtual datalink layer also enables simple adaption of other new networking technologies.

BACnet was from its beginning designed to allow interoperability of devices supplied by different manufacturers. Since for typical devices, only a subset of all available services is required, BACnet Interoperability Building Blocks (BIBBs) are defined. They define sets of services a device must implement to enable particular functional capabilities. On top of the BIBBs, BACnet device profiles, specify collections of BIBBs which correspond to the functionality of typical building automation device classes. Manufacturers may claim conformance of their devices to a given profile, if at least the included BIBBs are implemented. It is however still possible to add additional functionality.

To ensure interoperability between devices of different manufacturers, interoperability testing and certification programs have been established by both the BMA and BIG-EU.

3.5.3 EIB/KNX

EIB/KNX [25, 26, 19] is an open specification for home and building automation and part of the KNX standard, maintained by the Konnex association. EIB stands for European Installation Bus which was the original fieldbus standard which was later merged into KNX. EIB/KNX is intended to specify ways of how sensors and actuators can be interconnected in a home or building. It separates the supply of devices from the transmission of control information. The standard is in principle open. To receive all necessary rights and resources to manufacture EIB/KNX devices, a company must join the Konnex Association which requires the annual payment of a membership fee.

The EIB standard was originally created by the European Installation Bus Association (EIBA) in 1991. It was successful and in wide use especially in German speaking and northern European countries. In 2002, EIB, Batibus and EHS (European Home System) were merged into the KNX standard, with the goal to create a single European home and building electronic system standard. After being accepted as an European standard (EN 13321-1 and EN 13321-2), in November 2006 the KNX protocol, including all transmission media (TP, PL, RF and IP) was approved for publication as the ISO/IEC 14543-3-x International Standard.

EIB was designed to enhance electrical installations in homes and buildings. Its initial application areas were lighting, shading and HVAC but the design is

powerful enough to fulfill complex automation tasks with tens of thousands of devices and over a large areas. A decentralized architecture was chosen to provide failure resistance by design. The standard was intended to offer a complete solution for the interconnection of home automation devices, from the physical data signaling in the field level to the high-level protocol parameters in the application-level.

In EIB/KNX up to 254 devices may be connected to a physical *line* in free topology. Up to 15 lines may be connected to a *main line* via a router, in this context called *coupler*, to form a *zone*. The lines may be arranged in a tree structure, loops are not allowed. Finally up to 15 zones may be coupled by a *backbone line*. Therefore, the maximum number of devices an EIB/KNX network can contain is 15 x 15 x 254 = 57150.

The levels of the hierarchy in such a network are usually mapped to the structural divisions of a building or site. Since the traffic on the backbone lines can be significant, often high speed connections like Ethernet (EIBnet) are used for these lines.

Every device contained in the network has a unique identifier, its *individual address*, which reflects the position this device has in the network. It is usually represented as a 16-bit number and contains the zone, line and device number. The individual address is only used for unicast communication, it is reserved for client-server style communication.

To enable a free configuration of sensors and actuators, so called *group addresses* are used. They allow the logical connection of devices in the network. For example, an actuator can be configured to react to messages sent to a certain group address. Devices can be associated with several group addresses, enabling flexible grouping and configuration of the network. Group addresses are defined globally for the whole network. However the specific location allows local and global group addresses by defining in each message the maximum number of Routers to be crossed.

The transport layer of the EIB/KNX network stack provides services for broadcast, multicast and unicast datagram transmission as well as reliable point-to-point connections.

In EIB/KNX a shared variable model is used for exchanging data between devices. The functionality a device implements is offered by *group objects* which can be read or written (or both) by other devices. Each group object must have a distinct group address assigned. The data exchange happens in an event driven manner. As soon as a data source changes its value, it is reported by pushing the new value to the network with the assigned group address. Since messages sent to group addresses are transported to all devices in a network, the knowledge of a group address is sufficient for a listening node to gain information about value changes. Additionally, there are also mechanisms to query group objects.

In addition to group objects, *system interface objects* allow accessing information like system management data or the loaded application program. *Application interface objects* offer user application related data, including fixed application parameters as well as run time values. The information in interface objects will typically be accessed by a PC-based tool or controller.

For the communication between nodes in an EIB/KNX network, several transmission media are standardized. Most common is the use of a twisted pair cable, which is known as KNX TP1. It is operated at a voltage of 29 V for signaling which may also power connected devices. The maximum length of segments using TP1 cabling is 1000m, up to four segments can be concatenated. The used data transfer rate is 9600 b/s, which is well enough for several thousand devices, if used correctly. To regulate the concurrent access of several devices to this common bus line, bit-wise arbitration and priorities are used.

Other media which are specified for EIB/KNX include:

- Powerline communication
- KNX-RF for wireless communication
- EIBnet
- KNXnet/IP

EIBnet transmits EIB messages using Ethernet technology, basically enabling the fusion of EIB/KNX and Ethernet LAN networks. KNXnet/IP further allows the extension of EIB/KNX lines over IP networks.

EIB/KNX not only specifies the protocol for communication but also some standard system components. There are three types of specified devices:

Bus Coupling Units (BCUs) implement a complete network stack and application environment. BCUs are intended to provide a mounting platform for application modules but may also host small application programs themselves. They are often flush mounted in electrical installations.

Bus Interface Modules (BIMs) enable the communication with an EIB/KNX network. BIMs are functionally reduced BCUs without casing, allowing tight integration into devices.

TP-UARTs (Twisted Pair - Universal Asynchronous Receive Transmit) are integrated circuits with the sole purpose of connecting devices to EIB/KNX TP1 cabling. They are fed via a UART interface and translate received messages to EIB/KNX bus telegrams, taking care of timing and voltage levels. TP-UARTs allow the highest level of integration into a device, but require the most engineering effort since no network stack is provided.

KNX is designed as a vendor-neutral standard, allowing products which adhere to the standard to interwork in the same system, even if they are from different vendors or application fields. This approach has several technical and commercial advantages which contributed to a large extent to the success of KNX. Such an interworking is made possible by the definition of functional blocks, which specify data types, data points and communication mechanisms to be used. KNX defines such functional blocks for a whole set of functions, covering typical building automation tasks.

To guarantee the interworking ability of devices, the KNX Association runs a certification scheme for products. The KNX Association's Certification Department is responsible for managing the certification process and granting the KNX logo. For testing of the products, a number of KNX accredited test labs can be used.

4 Approaches to improve security

There is a broad range of approaches which target at improving application level security in the presence of programming flaws or untrusted code, from static and dynamic software to hardware supported methods. Such methods have one thing in common: they are never able to offer full protection and often human preparation or interaction is required to make them useful.

In this section, research related to software attack countermeasures is presented. Several general techniques are introduced along with some existing methods and tools.

The presented methods are categorized into static software methods, dynamic software methods and hardware supported methods as suggested in [27]. Further categorization is used where appropriate.

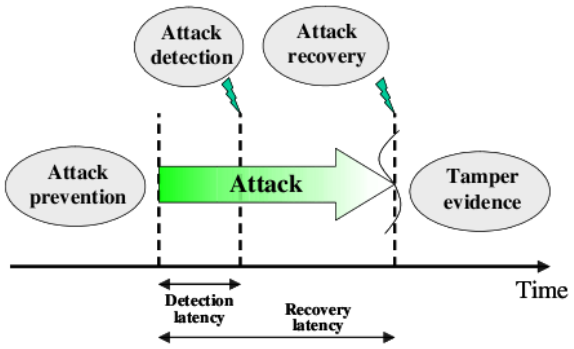


Figure 10: General stages of an attack [6].

The list of techniques presented does not claim completeness. Considering the attack model depicted in Figure 10, the focus is on techniques which address attack prevention and attack detection. In some cases, the focus was further narrowed on methods related or possibly applicable to embedded systems.

4.1 Static software techniques

4.1.1 Static analysis

Static code analysis (SCA) generally refers to analyses of program code to detect certain properties of a program without executing it [28]. In security, it is used to detect programming flaws that result in vulnerabilities. This includes manual as well as automated checking using special tools.

Manual checking by humans requires extensive knowledge by the auditing person. A programmer usually does it automatically while programming but is often unable to find some of his faults. Manual checking by other persons in a review fashion can be very effective in eliminating flaws but is highly time consuming and thus expensive.

Automated tools use pattern matching to detect common program flaws. This may be as simple as using a program like grep to find usages of dangerous functions in a set of source files. More sophisticated tools combine techniques like annotations, heuristics and modeling the execution state. Commonly possible programming flaws are identified and printed in a human readable form. Such tools can be well integrated in the development process and may provide valuable feedback to the programmers.

Automated SCA can be carried out on program source code as well as on already compiled binaries. If carried out on compiled binaries, decoding the instructions can be difficult. Especially in the area of embedded systems, varying instruction set architectures hamper the use of general binary code SCA tools. On account of this, most recent publications on binary static analysis focus on the Intel x86 architecture, like [29], [30].

A lot of SCA tools work on program source code, especially on low-level languages like C where programming flaws can easily result in exploitable security problems. Since the analysis takes place before the compilation process, it can be portable across different target architectures, although on the other hand, not every architecture is vulnerable to the same attacks.

A problem of SCA is that for languages with if-statements, loops, dynamic storage and dynamic data structures some fundamental questions are undecidable or uncomputable [31]. That means the results of an SCA tool are always only approximate and can never be sound and complete. Therefore it will always be uncertain if some of the detected flaws are false positives, or some real flaws remain undetected.

Available open SCA tools

In the following, a selection of available tools for SCA will be described.

Splint is a quite popular open source tool based on lint for statically analysing C source code. According to its development homepage (http://www.splint.org/), “Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as a better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint.”. The annotations that

can be used with Splint make it possible to explicitly add information about the intended behaviour of a program. They have been introduced in [32], and provide an interesting tool for experienced programs to write secure code.

ASTRÉE is a static program analyzer developed at the Laboratoire d'Informatique of the École Normale Supérieure (LIENS) [33]. It is intended to prove the absence of run time errors in C programs. ASTRÉE is able to analyze structured C programs with complex memory usages, although no dynamic memory usage and recursion is allowed. But even with these restrictions it is still able to analyze most programs written for embedded systems where such complex techniques are usually not present. ASTRÉE has been in use for several important applications, for instance in the field of aerospace.

Sparse is a semantic parser for ANSI C which also includes a static analyzer [34]. Originally started by Linus Torvalds in 2003, it is now maintained by Josh Triplett. It supports the use of annotations which can be used to include semantic information about data types or the intended use of function into the program.

4.1.2 Code signing

In *code-signing*, executables are signed by their producers to confirm the software author and to guarantee that it has not been altered or corrupted [3] . The user can on reception of the code check its correctness and determine the identity of the producer. With this information, the user can decide if the source of the code is trustworthy and therefore probably secure to execute. Usually, instead of deciding about the trustworthiness every time a program is received, the user manages a list of trusted entities or trusts a certification authority which does that.

Typically, a combination of a high quality hash function and public key cryptography is used for the generation of the signatures. Since public-key algorithms would be too slow for signing big documents, a hash value of the data is generated instead which is then signed and appended to the document. In addition to the hash, other information about the program and its producer can be added.

A problem with code signing is that it requires the user to have some common sense about whom should be trusted or not. This means, code signing does not provide automatic security for downloaded programs.

Somewhat similar to code signing, but with different purposes, is software *watermarking* [35]. It is used to embed secret (and intentionally non-removable) information into a piece of data, usually to assure that the rights of the creator are not hurt. A common application is media watermarking, where information about the owner and the copyright of a creation is embedded into it, for example a movie or a picture. Watermarking can also be applied to program software.

The injected watermarks are intended to be resilient to removal attempts (de-watermarking attacks), but also exhibit a trade-off between data-rate, cost and stealth.

4.1.3 Proof-carrying code

Proof-carrying code (PCC) [36] is a technique where a code producer provides a proof along with a program which allows the user to check, with certainty, that the code is secure to execute. For example, some programs can be proven not to contain any buffer overflows. To use PCC, the user needs to specify a set of rules that guarantee secure behaviour of programs. The code producer, on the other hand, has to create a formal proof which proves adherence to these rules. To check the proof correctness, the user is be able to utilize a simple and fast proof validator.

PCC has applications in many systems whose trusted computing base is dynamic. For example, it may be used in systems where the execution of downloaded code is allowed like in web browsers or embedded systems. As advantage to cryptographic code signing, PCC does not require any trust relationships between the code producer and the user. All information needed for determining that the code is secure is contained in the code and the proof.

If modifications have been applied to the code, PCC guarantees that there are only three possible outcomes when validating the proof:

1. The proof is no longer valid, the user can reject the execution of the program.

2. The proof is valid but does not correspond to the program anymore.

3. The proof is still valid.

In the third case, the program is still proven to be secure although the behaviour of the program may have been changed.

Although the concept of PCC sounds very promising, there are a number of problems which still need to be solved. The biggest source of problems is the generation and encoding of the proofs. The proofs can be checked efficiently by the user, but the generation is a complex task. Although automated proof generators have been developed, they have difficulties in generating proofs for all types of programs and against complex security policies. In [18] it is even stated that “there are properties related to information flow and confidentiality that can never be proved this way”. Another problem with the proofs is that they often have a significant size, which can be an order of magnitude larger than the code.

4.2 Dynamic software techniques

Dynamic software techniques for application security generally try to hamper software attacks by applying security mechanisms at runtime, either by hindering attacks a priori, or by detecting malicious or undesirable actions and reacting to them according to a given policy. A wide range of approaches fall into this category. There are techniques which are targeted at very specific attack conditions as well as ones which are supposed to provide general means of security.

While in the IT world, *operating systems* (OS) typically limit what an application is allowed to do, the targeted microcontrollers do not provide the necessary hardware support (e.g., memory management units to separate the address spaces processes can access are often lacking). Traditional OS for such lean ES thus cannot provide comparable protection or are not even designed to provide security measures. Moreover, in general trusted software such as an OS cannot be guaranteed to contain no flaws. For completeness, it has to be noted that possibilities exist to overcome the lack of memory management units [37].

Here, dynamic software techniques are organized in five categories: *intrusion detection*, *attack specific techniques*, *software monitoring*, *sandboxing* and *code signing*. Additionally, approaches which do not fit well into any of these categories are presented in the end of this section.

4.2.1 Intrusion detection

Intrusion detection systems (IDs) monitor the behaviour of a system and use the collected information to detect malicious modes or actions [38] [39]. There are several ways to collect this kind of information: tracing system calls, file system operations or network traffic are common.

Two general approaches for intrusion detection can be distinguished: *signature based* intrusion detection (SID) and *anomaly based* intrusion detection (AID).

SID systems rely on a collection of signatures describing known attacks. The signatures can for example describe parts of files, memory regions, system call sequences or more abstract attack representations. By matching these signatures to information collected at runtime, malicious actions can be detected. Most modern antivirus software use SID to detect malware.

SID systems have a high degree of accuracy and can be implemented efficiently. If an appropriate abstraction is used, SID systems are to some extent also able to detect new attacks which are similar to known patterns. But they are unable to detect truly novel attacks. Therefore the signature database has to be updated as soon as new attacks are discovered which may be difficult in some installations. SID systems are also vulnerable against attack variations, for example worms that change their own code.

The goal of AID systems is to detect unusual system behaviour or data which is assumed to be part of software attacks. This is done by comparing program behaviour or network traffic to a representation of "normal" operation. The principle is to some extent similar to the human immune system.

AID methods use models that describe normal behaviour. They are built upon feature vectors which contain parameters from various sources of information. The observed data is characterized as normal or abnormal, using mechanisms from simple statistical analysis to neural networks and other AI techniques. The models used in AID have to be trained with normal system behaviour to be able to distinguish it from anomalies.

AID is able to detect novel intrusions but suffers from high false alarm rates. It is difficult to distinguish attacks from natural changes in the system (e.g. different users, different applications). On the other hand, intrusive activities that appear to be normal remain undetected. Also, due to the uncertain nature of AID systems, it is problematic to apply them to high speed systems.

4.2.2 Software Monitoring

Software Monitoring techniques generally observe the execution of programs. By identifying and reacting to certain security relevant events they can check if programs behave according to a given security policy. On the detection of malicious behaviour, actions can be taken to prevent it or to stop the program. The used policies can be defined by a security expert or automatically generated using static analysis of the monitored programs. Monitoring usually takes place at instruction level by checking some or all instructions which are to be executed by a program.

Program shepherding by Kiriansky et al [40] specifically monitors control flow transfers in a program and prevents transfers to data or modified code regions. This is done by verifying every branch instructions a program is about to execute to ensure that each satisfies a given security policy. To this end it is possible to specify a security policy at a fine-grained level, allowing the definition of restrictions regarding code origins as well as control transfers. To make sure these restrictions cannot be ingnored a technique called un-circumventable sandboxing is used.

Janus, by Goldberg et al [41], was built as secure environment for untrusted helper applications. They identified helper applications (e.g. in browsers) as source of security concerns. Therefore the hosting application is secured by intercepting and filtering dangerous system calls. Janus, still under development, runs in user-mode and can be applied to pre-existing applications.

4.2.3 Sandboxing, Virtualization

According to a definition by Tanenbaum [2], "a sandbox is a technique by which a downloaded program is executed in such a way that each of its instructions can be fully controlled". A sandbox is often used to execute untrusted programs or untested code. The essential benefit of executing code in a sandbox is that the system outside the sandbox is protected from malicious actions by the user application. Additionally, the behaviour of the program in the sandbox can be monitored and controlled.

As an extension to sandboxing, the concept of a virtual machine (VM) is used to execute so called *applets*. Applets are self-contained programs which run in the context of other programs (e.g. web browsers).

Virtual machines can be classified into *process* and *system* VMs [42]. Process VMs apply their virtualization mechanisms between the operating system and processes. Most modern operating systems implement similar mechanisms by supporting multiple user processes. This way, a process can be given the illusion of having the whole machine to itself. In system VMs, the virtualization software is situated between the operation system(s) and the hardware. This allows several operation systems to coexist on the same hardware.

Today the term sandbox is often associated with the Java runtime system, where it is an essential part of the security architecture. The Java sandbox model can be described as a collection of mechanisms [43]:

- Java class files are verified for correctness.
- Only a restricted API is available to untrusted code.
- Class loading mechanisms are part of the VM and cannot be bypassed.
- The native functionality of the VM is closed.

The sandbox model in Java also evolved over time and contains several advanced security mechanisms to date. For example, a security manager class checks and restricts actions performed by untrusted code. An advanced class loader ensures that untrusted applets cannot interfere with the operation of other java programs.

Sandboxing in embedded systems

Especially targeted at mobile embedded systems, Scylla [44] is a is simple, fast and robust virtual machine architecture It offers the capability to run code compiled for a common instruction set architecture (ISA), independently from the underlying hardware. The basic instruction set is closely matched to popular

processor architectures to allow efficient on-the-fly compilation. Additionally, instructions for inter-device communication, power management and error recovery are provided.

There are also a number of Java virtual machines available for embedded systems. Sun offers the official Java Micro Edition (Java ME) [45] for embedded devices, targeted at mobile phones, personal digital assistants (PDAs) and similar. Although strictly reduced in size and overhead, it still features powerful programming interfaces, robust security and built-in network protocols. But due to this relatively comprehensive feature set, the minimum system requirements specify the need for 160-512 kilobytes of total memory and a 16 bit processor clocked at 16 MHz or higher. Especially the memory requirements are slightly beyond the limitations of low-end embedded systems which were set in Section 2.5.

Besides the official Java versions, there is a number of implementations available which are targeted at systems with very low resources. These implementations often come at the price of only offering a subset of the full Java functionality. Only two such solutions shall be presented here: NanoVM [46] and TinyVM [47]. Both virtual machine implementations allow the execution of standard Java bytecode, at least after preparation with an automated tool. They target very low-profile embedded systems. The NanoVM was originally written for the Atmel ATMega8 microcontroller included in the Asuro robot, and has a memory footprint of approximately 7 kilobytes. The TinyVM operates on Lego Mindstorms RCX programmable bricks which are equipped with a Hitachi H8 microcontoller. It has memory requirements of around 10 kilobytes. Both VMs have some limitations regarding the provided Java language features. For example the NanoVM does not support multithreading. The TinyVM is generally more advanced (of course also due to the more powerful hardware it runs on), and supports multithreading, exceptions and synchronization but for example misses floating point operations. But despite these limitations both VMs have their possible applications, since the omitted functionality is often not required for their intended uses.

4.2.4 Self checking code

Self checking code (SCC) usually describes techniques where programs check themselves for modifications. Assuming that modifications are unwanted and probably malicious, software attacks can be detected in this way. Static and dynamic self checking code can be distinguished. Static techniques check the program integrity once before the program starts while dynamic techniques do so at runtime.

A simple approach for SCC would be to generate hash values over parts of the program and compare them to stored values, usually determined at compile time. This way, changes in the program code can be detected by regenerating

these values at runtime and comparing them to the stored values. One drawback of this simple approach is that the added SCC mechanisms can easily be distinguished from normal program code and could be bypassed by an attacker. Therefore, measures have to be taken to hamper the discovery and modification of the included SCC mechanisms in order to make them effective. For example, SCC could be used in conjunction with watermarking, obfuscation and anti-debugging techniques.

Horne et al [48] describe a technique where several so called *testers* and *correctors* are injected into programs. Every tester has an assigned block of code that it generates a hash value of. Correctors are inserted into these blocks to make the testers hash their intervals to a fixed value. This approach is claimed to be more secure than storing the expected hash value in the testers.

Oblivious hashing by Chen et al [49] is a somewhat similar approach but it has a special focus on the obfuscation of the checking mechanisms. They state that it is very easy for an attacker to identify and in succession disable or bypass normal checking routines, since they are quite different to common program code. Programs usually do not read their own code sections. Therefore they introduced a technique that continuously calculates a hash value based on the dynamic execution context of the host code. This way operational correctness can be checked at any point in the program. The main benefit of this approach is that the checking code can be blended seamlessly into the application code, since is uses similar instructions and is therefore hard to detect.

4.2.5 Attack specific counter-mechanisms

There is a number of security attack types which often exploit vulnerabilities in a similar way. For example, buffer overflow attacks usually overwrite parts of an application's memory to change its behaviour, often to gain some sort of control over the hosting machine. Format string vulnerabilities rely on the insecure uses of C format strings, used in the `*printf` family of functions of the standard library. In such cases, the assumed attack type requirements are narrowed to a set of conditions, and mechanisms to mitigate them are installed.

On this account, several approaches exist which focus solely on protecting against specific attacks. StackGuard [50] was an early method to hamper buffer overflow attacks. It applied several techniques to detect or prevent buffer overflows, for example storing return addresses for comparison or placing known values (canaries) between structure variables for detecting overwrites. Thereafter some similar but more advanced methods like StackShield and ProPolice evolved. The development of FormatGuard [51] was a response to the relatively new format string attacks. It includes function argument counting, replaces the printf() function in libc with a more secure implementation. Similarly, RaceGuard is an

approach to circumvent the exploitation of race conditions.

Such approaches can be effective in some cases but since they use some assumptions on the attack type, they cannot always catch all attacks. For example, in a comparison in [52], even the best buffer overflow prevention method only works in 50% of all tested cases. Furthermore, since the inner workings of such methods are usually publicly available, attackers can easily try to find ways to bypass them.

4.3 Evaluation

Method	compile-/ runtime	Attack types	Updates required	Applicable to low-end ES
Static Code Analysis	ct	known	yes	+
Code signing	ct	/	no	$\sim$
Proof Carrying Code	ct	all	no	$\sim$
Signature based ID	rt	known	yes	-
Anomaly based ID	rt	all	no	+
SW Monitoring	rt	all	no	+
Sandboxing	rt	all	no	$\sim$
Self checking code	both	all	no	$\sim$
Attack specific	both	specific	no	$\sim$

Table 1: Comparison of described software methods to improve application security.

In Table 1, an overview of all described methods is given. Every method is evaluated in terms of applicability to low-end embedded systems software. The evaluation presented here is to a large extent only the opinion of the author.

Static Code Analysis is assumed to be easily applicable because it is only used at compile time and does therefore not use any resources of the target system. The same argument can in principal be used for Proof carrying code, although it may be questionable if it can be of use in securing an embedded system. It could be useful, for example, when programs are written by an untrusted party and the user wants to make sure that the program does not contain any backdoors. Code signing could be quite effective to prevent the installation of malicious programs by simply refusing to execute not properly signed ones. On the other hand, checking the digital signature is a quite complex operation and may overextend the processing power of an embedded system. Also, the public keys of the trusted source must not be allowed to be overwritten.

Signature based ID is not well suited as it depends on a usually large database and requires constant updates which would be difficult for many embedded sys-

tems installations. Anomaly based ID, Software Monitoring as well as Self Checking Code may be efficiently implemented and could therefore be quite appropriate. Attack specific mechanisms could also work well, but are not generally applicable due to differing processor and memory architectures. The applicability of Sandboxing strongly depends on its feature set. While a basic sandbox could easily be deployed, an architecture like the Java VM with its vast execution and security mechanisms imposes a big overhead.

4.4 Hardware supported techniques

There is one problem common to all dynamic software mechanisms for improving security: they are usually executed on the same underlying processing hardware. Therefore, the mechanisms are often exposed to security attacks themselves. Mechanisms implemented in hardware are certainly not secure by definition either, but they may pose an extra barrier to attackers that is not easily bypassed with conventional techniques.

There are several approaches that use hardware mechanisms to provide security enhancements. A common technique is to attach a coprocessor to the processor executing the program which performs security checks at runtime, like described in [53], [27]. This often involves static analysis of the executing programs, to e.g. generate call graphs that can at runtime be compared to the monitored program behaviour. The coprocessor needs to be closely attached to the monitored processor to enable extensive low-level monitoring.

The increasing use of multicore architectures allow their usage for security mechanisms. Physical partitioning [54] may be used to improve a systems reliability as well as its resistance against security attacks.

Processors implementing the Harvard Architecture provide resistance against code injection attacks by design. The separation of instruction and data memory makes it impossible to execute injected code. Riley et al propose a change to the memory architecture of modern processors to imitate the Harvard architecture on modern von Neumann processors. The want to split the memory virtually into code and data memory, therefore a processor would never be able to fetch injected code execution. In their latest paper [55], they even introduce a software only patch which could be applied to operation systems running on x86 architectures.

A recent security technique is the no execute (NX) bit, introduced by AMD [56]. It is now present in most modern processor architectures, only with different names, e.g. Intels XD bit (eXecute Disable). It allows memory regions to be designated as being non executable. That means they may only be used to store data making traditional code injection attack like exploing buffer overflows impossible.

5 Proposed architecture

As an approach to provide comprehensive security in low-end embedded systems, a secure architecture for such systems is proposed in this section. It is intended to deal with the problems described in Section 2.5 by applying some of the techniques described in Section 4. The focus is on the resistance against software attacks, while hardware and side-channel attacks are only considered to a little extent.

The main goal of the architecture is to create a secure software architecture which allows uploading and executing arbitrary, uninspected and uncertified (and possibly erroneous or malicious) software without compromising the overall system security. Not only the security of the hosting system but also of its environment, like its surrounding network, is to be protected. At the same time, sufficient flexibility for a wide range of applications should be provided.

It is important to note that not only problematic behaviour evolving from accidental software faults is tried to be prevented but also attacks resulting from intentional malicious user applications. The assumption is that an attacker who has full control of the user applications is trying to harm the system or its environment. The architecture should guarantee that under any circumstances, he is unable to do so.

The solution shall be efficient and not rely on any special hardware modifications, thus allowing easy integration into existing embedded networks. It should require only a proportional amount of resources to enable its application in low-profile systems. Additionally, it should provide a decent usability, meaning that writing applications for such an architecture should remain simple and not require the use of adapted programming techniques.

5.1 Concept

The idea is to reasonably combine the advantages of different methods presented in Section 4 to create a system software which is resistant against software attacks. As outlined in Figure 11 the presented architecture uses three mechanisms to increase this resistance: Static code analysis, anomaly based intrusion detection and a sandboxing solution for controlled user application execution. Each of the adapted mechanisms imposes an additional security barrier to the overall security and limits possible attack points:

Static code analysis: A simple, tight and secure system software provides controlled access to system resources. Its security is analyzed using inspection, code reviews as well as automated SCA using software tools. This long-lasting process has to be done very thoroughly since mistakes in this stage

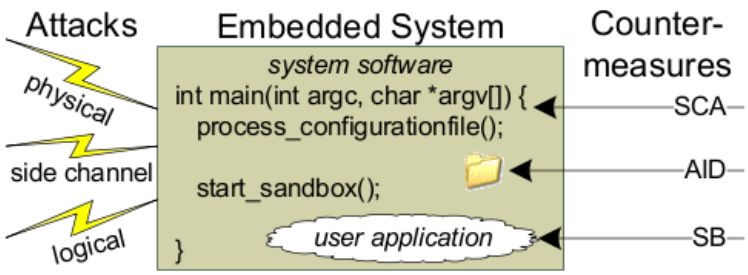


Figure 11: Secure software architecture.

may easily squash any later efforts in developing a secure platform. However, this (extra) effort is not for nothing since such an established common system software for a particular architecture/processor, may – once considered secure – serve as a common code base for any other ES.

Sandboxing: A sandbox restricts the execution of a customizable user application. This uploadable code shall, in addition to basic operations like calculations, only be allowed to perform a defined set of actions. The set needs to be well chosen to allow the user applications to carry out their desired functionality on the one hand but on the other hand prevent it from doing any damage to the system and its environment. The actions could be defined in the form of a restricted software library to be used by the user application. Its methods may depend on the desired application area of the system.

Execution of applications could be further limited to those being signed and containing valid cryptographical signatures (CS) to provide support for DRM.

The sandbox may also be designed to support the rapid development of applications. A clear abstraction of the underlying hardware and interfaces to the system software could be provided. The application designer would then be relieved of any hardware or device specific details and could focus on the application development itself. This would allow portability of applications between devices offering the same sandbox.

Invocation monitoring: An AID-like protection mechanism is used which monitors the execution of the user application and compares it to a given policy. It is necessary because allowing the user application to only invoke a restricted set of actions (through sandboxing) may not always be enough. Some methods which are needed for delivering normal functionality may be used to harm a system or its environment by invoking them too frequently

(DoS attacks). For example, if a method to send a message is invoked continuously it could impede bus communication. Therefore, a set of rules defining the maximum invocation of certain methods is used. This way, objectionable invocations can simply be ignored or more serious reactions may be taken. Halting or restarting the system would be thinkable or, in more advanced implementations, a policy violation could be reported to a centralized management entity.

It is important to note, that the set of rules forms a major part regarding security since it allows to enforce complex security policies if properly designed. Obviously, an application designer has to provide reasonable values along with an application. However, it is always possible for the user to deny the execution of an application if the rule set is not restrictive enough. Besides, it is possible for a wide range of ES to define generic device profiles, which may be shared among applications of the same purpose. In such a way the device class of sensors may share a single profile with generic limits and the application designer does not have to provide an individual configuration. Instead, the rule set could be fixed for a specific device class.

Summarizing, the overall security of the proposed architecture relies on

- the system software not containing any flaws (e.g., buffer overflows) and not being modifiable to the user applications,
- the interfaces to the system software being restricted and being non-bypassable, and
- the system software being able to determine during runtime, whether the behavior of an application is malicious.

5.2 Architecture

How such a secure architecture could be realized is outlined in Figure 12, which depicts the combination and interaction of its related components.

The user application is run in a secure execution environment, the sandbox. It is executed in a controlled way, where every operation may be reviewed before it gets processed. The sandbox has to be designed in such a way that an application inside it cannot randomly access memory regions or resources outside it. Such access should exclusively be possible through a restricted user API. For all these demands, a virtual machine, interpreting application code, would probably be the best solution. It offers surpassing control over the execution of an application and can be used to restrict functionality to an arbitrary level.

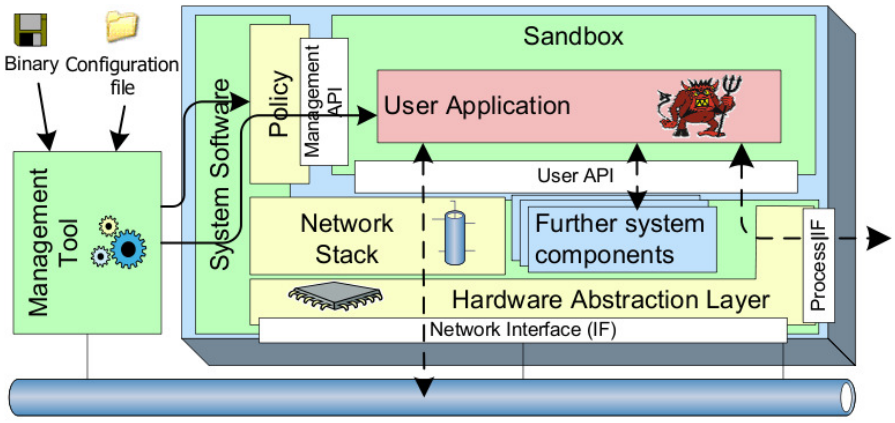


Figure 12: Outline of the proposed architecture.

The user API serves as an interface between the system software and the user application and provides various secure services (e.g., networking, access to on-chip peripherals such as timers, process interaction). Besides allowing to limit the possibilities of a user application such a generic API also supports portability and compatibility of applications on different platforms. The demands on the user API will of course vary for different application fields. Instead of trying to provide a general API to fulfill a wide range of demands, a specialized version could be composed for each application field. This would save space and further limit the number of actions a malicious application may abuse.

In addition to invoking certain services, a user application often has to be able to react to external influences. For example, it may respond to an incoming message or execute certain actions on pressing a button. Therefore, external events have to be detected and signaled to the user application via the user API.

In the ideal case, the user API should operate at the highest level possible. For example, in the case of networking, only valid incoming messages shall be reported to the user application, while erroneous receptions will automatically be discarded and negatively acknowledged. This way, the user applications could be kept simple and at the same time have less possibilities to perform unintended actions.

The system software is the glue between all participating components of the architecture. Besides taking care of initialization tasks and managing the available resources of the system, it...

- ...runs the sandbox, manages its required memory and provides an interface to the environment.

- ... implements the invocation monitoring of the user application being executed in the sandbox. Therefore it manages a set of rules to compare it with the monitored behaviour.

- ... provides a network stack and libraries for accessing further system components, for example for controlling peripherals.

- ... provides an external interface, the management API, through which it is possible to upload the user application and related configuration.

To access the hardware in an independent and modular way, a Hardware Abstraction Layer (HAL) is used. It allows simple utilization of the developed software on different hardware architectures enabling flexibility in design and to fulfill the differing resource requirements of ESs.

The management API interfaces with a management tool, which allows access to the system software to support the total replacement and download of the user applications. For the API to be secure, it has to be very restrictive and allow only minimal capabilities for possible attackers to directly access resources.

The hardware platform running the architecture should be chosen to further support the overall security of the system. For this reason, a processor implementing the Harvard architecture would be a significant benefit. In such processors the memory for instructions and data are separated, making code injection attacks impossible. A number of currently available microcontrollers implement the Harvard architecture. Resistance against physical and side channel attacks may also be considered when building the hardware platform. For this reason it should for example be avoided to store any confidential information on an external storage since buses connecting them can often easily be eavesdropped.

5.3 Intended advantages

The following list describes the intended advantages the proposed architecture provides:

- User applications can only issue a defined set of operations.

- User applications have no access to low-level functions or random memory locations.

- The operations can be limited in terms of issuing frequency.

- Information invisible and unaccessible to the user application can be stored and used on the system (e.g. cryptographic keys).

- User application development is simplified.
- Portability of user applications can be achieved.
- It can be ensured that all (bus-)communication is standard compliant.
- Vulnerabilities caused by incalculable side effects are minimized by the use of static code analysis.
- Code injection attacks are impossible if a Harvard architecture is used.

Maybe the most important benefit is the execution of the user application in a controlled sandbox. While this approach may at first seem inappropriate for a low-end embedded system, due to the relatively high resource requirements of such techniques, it offers outstanding possibilities. Besides, the resource requirements can be lowered to a significant extent with the acceptance of certain limitations. The sandbox does not need to support fully fledged programming models, since the desired operations, especially in controlling tasks, are often quite simple. For such purposes, user applications more or less consisting of a sequence of simple operations may be sufficient which can be supported by a resource-saving sandboxing implementation.

6 Proof-of-Concept implementation

To demonstrate the flexibility and usefulness of the concept proposed in Section 5, a Proof-of-Concept (PoC) implementation was developed as part of the thesis. It follows the proposed concept as close as possible while using only minimal and cheap resources as a basis. While some limitations have to be accepted in terms of speed and stability, the present PoC delivers some interesting and promising results.

The PoC was intended to represent a low-end device of an HBA system. The goal was to be able to carry out HBA control tasks while offering a high degree of security against malicious application software. Additionally, the solution should offer decent flexibility with an adaptable configuration. The system software running on the device should be separated from the application programs and the node configuration. Ideally, the system software should not be alterable. On successful integration of the device, a test application was ought to be written to demonstrate the capabilities of the system and get a feeling for its characteristics. EIB/KNX was chosen as the HBA standard as it is widely spread in European countries and extensive documentation is available. Additionally, a suitable test environment for EIB/KNX was seizable in the A-Lab of the Institute of Computer Aided Automation (TU Vienna).

A custom hardware platform was designed to resemble SAC-like devices using only standard low-cost components. As the focus of the thesis is set on low-level embedded systems, a microcontroller with very little power consumption and low processing power was chosen. In addition to the basic circuiting required for the operation of the microcontroller, EIB/KNX bus connection was established using a slightly adapted version of the basic circuit of the Freebus project [1]. It allows EIB/KNX bus interaction with only a few basic components. Inquiries about its usefulness in real life applications were a small side goal of the thesis.

As basis for the software, a freely available virtual machine which allows simple modifications and extensions was to be chosen. NanoVM, a heavily reduced implementation of a Java virtual machine for the Atmel AVR processor family, was found to fulfill the stated requirements very well. It is developed by Till Harbaum and its code is released as open source software under the GNU General Public Licence (GPL). It offers a complete software framework for writing custom libraries. A standard Java compiler can be used together with a provided conversion tool to prepare the application code for execution on the NanoVM. Application programs can be stored in non-volatile memory and uploaded via an integrated bootloader. This allows changing the application program without touching the virtual machine software.

Although the NanoVM is a very powerful basis for the PoC software, signifi-

cant modifications and extensions of the present software system had to be done to achieve the targeted level of functionality. Software libraries had to be written to access peripherals of the hardware platform and to enable EIB/KNX compatible data exchange in application programs. Especially for the latter, the developed EIB/KNX library posed several difficulties, for example, quite strict timing requirements had to be met. Another challenge was the decided goal to have a flexible solution for configuring the EIB/KNX node parameters. To have the possibility of changing the configuration during operation, it is stored in non-volatile memory and managed by the system software.

Since applications running on the PoC are only able to use the provided library methods for any kind of interaction with other devices, a certain level of security could already be established by reducing the provided library functionality to a minimum. Additionally, to further prohibit unwanted behaviour, an AID-like invocation monitoring mechanism was implemented.

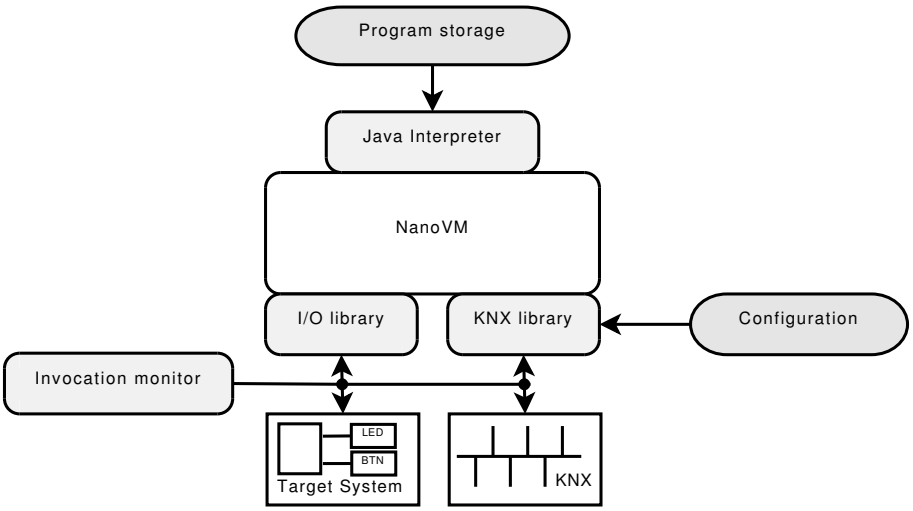


Figure 13: Sequence diagram of the PoC software components.

Figure 13 outlines the flow of information between the software components of the PoC. Java bytecode instructions are fetched from the program storage and are interpreted by the Java interpreter. The instructions may be operations on internal Java variables or calls to library methods. In the latter case, if methods of native libraries are called, they are being checked by the invocation monitor. Only if the method calls pass the checks, they get executed. The library methods execute their designated functions on the hardware platform or by communicating with peripherals. The KNX library also reads the configuration for some of its operations. Some events are only registered at the level of the library implemen-

tations, like pressed buttons or incoming messages, which are then signalled back to the application program.

The test environment was a small KNX network consisting only of the PoC, three pushbuttons and two lamps all connected to the same line. Two of the pushbuttons were configured to toggle switch the lamps. A desktop computer was also attached to allow comfortable bus monitoring using the Engineering Tool Software (ETS). Figure 14 depicts the test environment. The PoC was used in various test cases, for example to switch one of the lamps, listening to the pushbutton commands or combined interactions like switching a lamp when a command of a pushbutton was received.

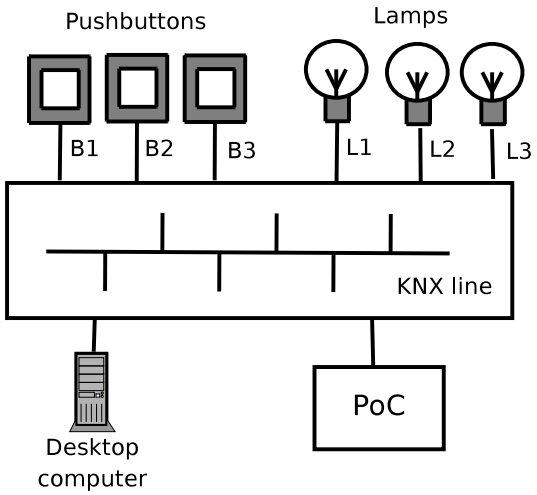


Figure 14: The test environment for the PoC.

In the following, the parts of the PoC are described in greater detail. This includes the hardware platform, the system software, the implemented libraries and the used tools.

6.1 Hardware platform

The hardware platform was developed and assembled as part of the thesis. It is mainly a collection of cheap standard components, equipped and soldered by hand. While it appeared reliable during development and testing of the software running on it, it is not intended for real-life use.

6.1.1 ATMega168

The core component of the hardware platform is an Atmel ATMega168 microcontroller, featuring 16 kilobytes flash memory, 512 bytes EEPROM and 1 kilobyte SRAM. Among others, the controller provides the following peripheral features:

- 23 programmable I/O lines.
- Two 8-bit Timer/Counters and one 16-bit Timer/Counter with various compare modes.
- A programmable serial USART.
- A byte-oriented 2-wire serial interface.
- Two external interrupts.

Additionally, it has a low power consumption while offering a decent throughput of up to 20 MIPS (at 20 MHz). In the PoC it is clocked at 8 MHz, the clock is generated using an external quartz oscillator. Power is supplied by an LM317T voltage regulator, the operating voltage is 3.3V in order to minimize the power consumption.

6.1.2 Peripherials

In addition to the necessary circuiting required for the operation of the microcontroller, some external components are equipped to provide additional functionality:

- For debugging, testing and basic interfacing an LED and a pushbutton are attached.
- An additional external EEPROM (MICROCHIP 24LC16B/P) with a memory size of 2048 bytes is used to provide extra non-volatile memory. It is connected via the 2-wire serial interface supported by the ATMega168, which is actually an I^2C connection (the naming is different due to licensing reasons). The connection is clocked at 400 kHz, which is the fastest possible with the external EEPROM. It still limits the data rate significantly, especially since there is some overhead contained in the required data exchange protocol.
- A MAXIM MAX 3232CPE dual channel driver/receiver chip is equipped to convert controller signals to RS-232 levels to enable serial communication, for example with a PC. On the side of the microcontroller, it is directly connected to the USART pins of the ATMega168.

For accessing KNX bus lines, a slightly adapted version of the basic circuit of the Freebus project [1] is used. Freebus is an open source project maintained by hobbyists which basically aims at providing a free and affordable system for home automation. The developers try to build devices with similar functionality as it is provided by retail manufacturers while using only a minimum number of parts to remain as cheap as possible. Additionally, compatibility with EIB/KNX bus systems is achieved to some extent. On top of the basic circuit, the project has already produced a number of powerful devices which are continuously improved. However for the PoC, only the basic circuit for the conversion of microcontroller signals to EIB/KNX signals and for the power supply using the bus line is used. Figure 15 shows its schematic as published on the project website at the time of this writing. This is also the version which was used for the realization of the PoC hardware platform.

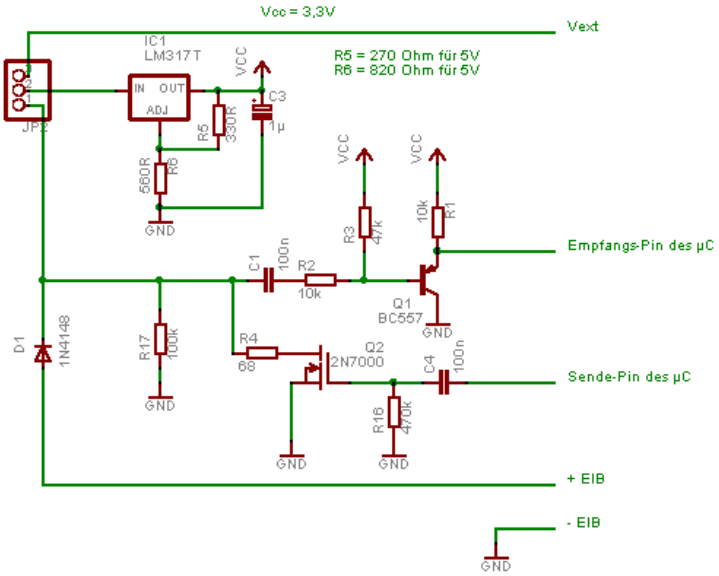


Figure 15: The basic circuit of the Freebus project. [1]

Figure 16 depicts the schematic of the hardware platform. Figure 17 shows the extension circuit with the MAX 3232 which was connected for serial communication.

6.1.3 Assembly

The hardware was assembled on a stripboard (2.54mm grid) using standard DIL components and wires for interconnections. The board was mounted on a lab panel for easy interconnection with other devices. Figure 18 shows a picture of

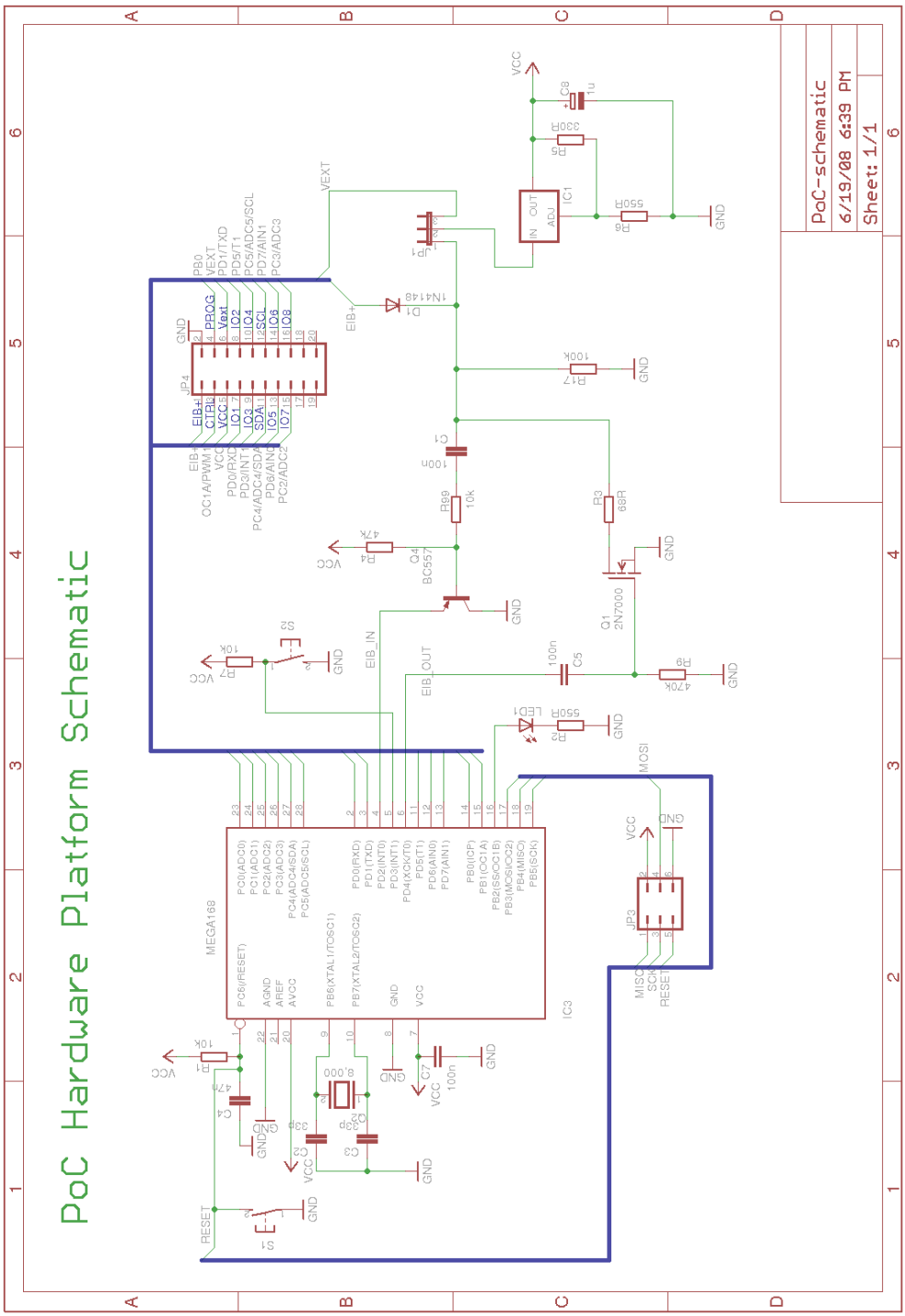


Figure 16: Schematic of the Proof-of-Concept.

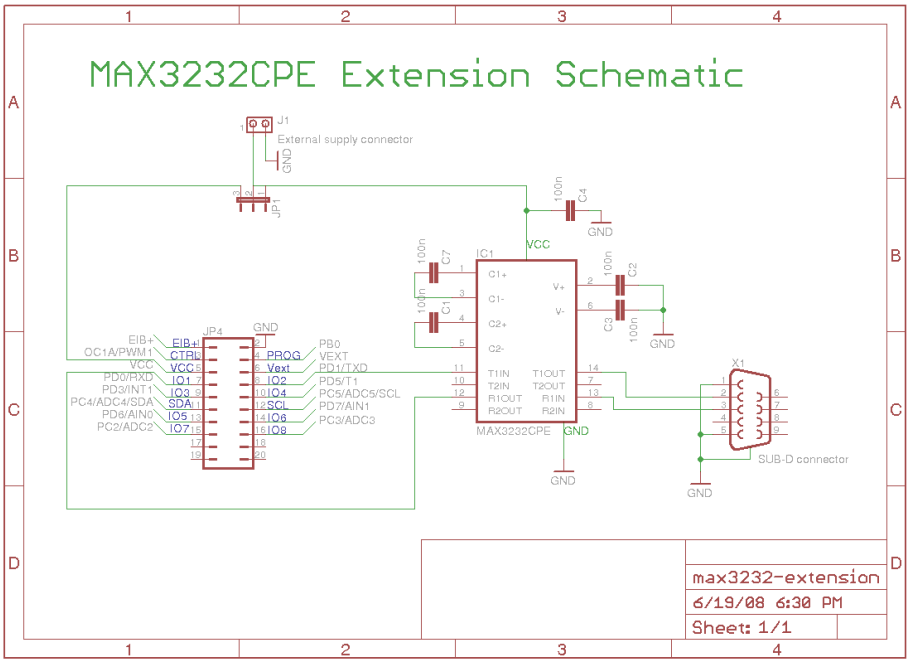


Figure 17: Extension circuit with the MAX 3232 for serial communication.

the assembled board. The ribbon cable which can be seen on the left hand side is used to interconnect the microcontroller to the MAX 3232. The pin assignment of this connection is compatible to the Freebus interconnection interface which can be seen in Figure 19. This makes it possible to connect application boards to the current controller circuit for more advanced applications.

6.2 Software

In this section the system software of the PoC will be described in greater detail. The software builds upon existing solutions with some significant extensions and modifications. The working title for the software solution is seBAS, an abbreviation of secure BAS. It is chosen to reflect its intended use as a secure solution for building automation systems.

Since all existing software used is released under an open source license, the PoC software is also planned to be released under an open license.

6.2.1 NanoVM

The core component of the PoC system software is based on the NanoVM, a Java interpreter designed which is able to run specially prepared Java bytecode. It is

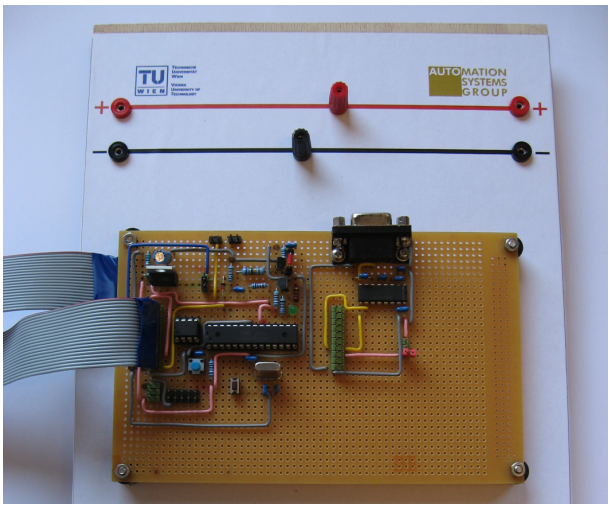


Figure 18: A picture of the Proof-of-Concept target system. The connectors on the top can be used to connect to an EIB/KNX bus, the board itself is wired underside.

targeted at embedded platforms and was developed to run on the Atmel AVR CPU family. It is being developed by Till Harbaum [46] and released under the GPL.

NanoVM is written in C and designed to be portable and extensible. Currently, it runs under Linux, Windows, some AVR CPUs and offers some specialized functionality for small robots. It provides Java interface classes enabling applications to control target hardware features like ports and timers. On an AVR controller, the NanoVM is intended to reside in volatile flash memory while the application Java program has to be put in non-volatile internal EEPROM. This allows changing the application program without touching the system software. The NanoVM original memory concept is depicted in Figure 20.

Among others, NanoVM includes the following features:

1. Configurable 15/31 bit integer arithmetic
2. Optional floating point support
3. Garbage collection
4. Boot loader for simple application upload
5. Support of inheritance mechanisms
6. Unified stack and heap architecture
7. About 20k Java opcodes per second on 8 Mhz AVR

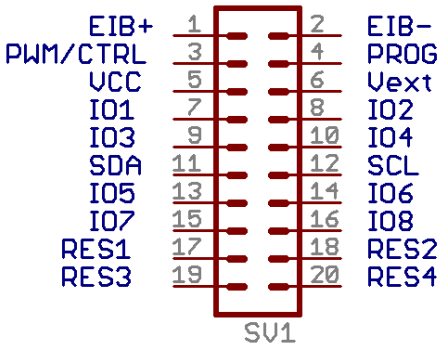


Figure 19: The Freebus interconnection interface.

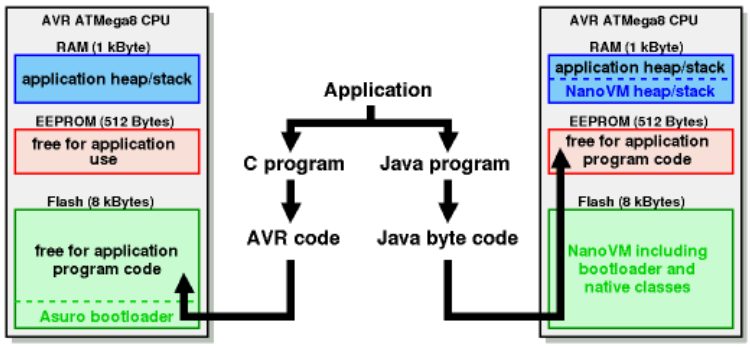


Figure 20: The original NanoVM memory concept. [46]

It has to be noted, however, that the NanoVM cannot be considered as a full-fledged Java VM. Basically it is just an interpreter of bytecode instructions. Since the code size of the NanoVM is targeted to be as small as possible, it lacks several mechanism of a full blown Java virtual machine. It does, for example, not support exceptions, multithreading and callback methods. Especially the latter is quite limiting as it hinders external events recorded by the processor to be reported to the application program in an asynchronous way. Such event information has to be polled by the application program.

The current version (1.4) compiled for an AVR ATMega168 results in about 7 kilobytes code memory and uses only 256 Bytes application RAM. All available EEPROM is used to store the Java programs. Thus 9 kilobyte code memory, 768 bytes RAM and 512 bytes EEPROM remain free on an ATMega168.

As mentioned before, the NanoVM runs only specially prepared Java byte-code. An application program can be compiled using a standard Java compiler

but it must be prepared using the provided NanoVMTool before it can be loaded by the NanoVM. These preparations include stripping unnecessary and unsupported instructions from the binary as well as mapping native Java library calls to their corresponding implementations in C. The tool is written in Java. In addition to the required preparation, using the tool has several advantages:

- The total file size is reduced.
- All depending class files are combined into one big file.
- Code optimizations can take place.

Considering the security of the NanoVM, since the code interpreter misses certain boundary and validity checks due to performance reasons, it can be crashed by intentionally prepared irregular code. However, there is no way that a malicious application program can inject code, damage or take over control of the system software. Partially, this is ensured by the used AVR microcontrollers which implement the Harvard Architecture.

Modifications

For its use with the PoC, considerable modification had to be applied to the NanoVM. Since the internal EEPROM of the AtMega168 was ought to be used for the EIB/KNX node configuration, the program storage had to be moved to the external EEPROM. For this purpose, a C library was created for accessing the contents of the external EEPROM in a comfortable way. This library was then used in the NanoVM program file implementation to load instructions. Although this approach resulted in a significant slowdown in the fetching of instructions, it had the benefit that the available program memory was quadrupled from 512 bytes to 2048 bytes.

As indicated before, it was intended to place certain software components of the PoC in memory types with different access modes. Figure 21 depicts the resulting memory concept. It shows the used memory types in general, the memory types of the implementation and the software components matched to their designated locations. Additionally, the parties who are allowed to modify them are indicated.

The invocation of native library methods had to be modified to allow monitoring of the invocation frequency. Basically, before a native method is invoked, it is checked if its method identifier matches a set of predefined security rules. If it matches, the invocation is recorded and it is verified that it does not exceed one of the limits defined by the rules. Otherwise, certain predefined consequences follow. In addition to checks at the time of invocation, a timing routine is also required to measure the timespan between the method calls.

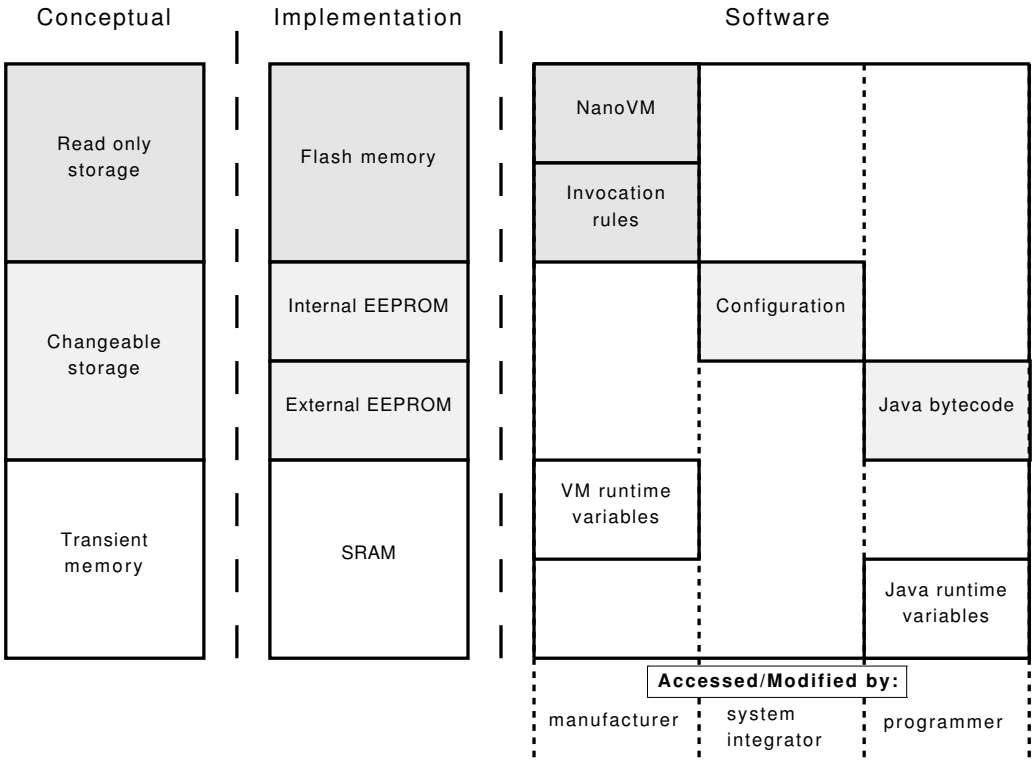


Figure 21: Overview of the PoC memory layout.

6.2.2 Implemented Libraries

Native Java libraries had to be written to provide the PoC with the needed functionality for its intended usage scenario. The NanoVM provides an easy and well documented way to do so. Classes are simply defined in Java by specifying all desired methods and variables. Method implementations have to be omitted. Instead a mapping between the Java methods and numerical identifiers has to be defined. These identifiers are then used in the C implementation to determine the intended method invocation whereupon the corresponding C code is executed.

In the following subsections the implemented libraries are described in a general way. A reference of its included methods and variables is given in Section 6.3.

seBAS I/O library

A basic I/O library was written for accessing the peripheral components of the hardware platform. It allows fetching the state of the attached pushbutton as well

as controlling the LED on the hardware platform. Additionally, a method to wait a certain timespan is provided.

EIB/KNX library

A much more complex library was developed for allowing EIB/KNX bus interaction in application programs which can be used for comfortable reception and transmission of messages. For this purpose, an EIB/KNX network stack was written. Although the Freebus community provides a similar stack, it was not used but rather written from scratch.

The library offers two interface modes:

1. A high-level mode where communication is done by solely reading and writing group objects. Messaging related tasks are carried out by the library implementation.

2. A low-level mode where almost the entire messages can be accessed in a byte-wise order. In this mode, message composition and handling has to be done by the application program.

The desired mode can be chosen by the application program. Of course the mode can also be fixed in the system software at the time of compilation if, for example, low-level processing should be forbidden. It is also possible to use the low-level mode as fallback for messages which cannot be automatically processed by the library. This is for example the case if the received message does not contain a standard KNX command or if the message is addressed to the individual address of the node instead of a group address. In such a case, the reception is signalled via the low-level interface if the fallback option is set. The user application may then process the message byte wise. Without fallback, such messages are simply discarded.

The actual message transmission is taken care of by the library. This includes checksum generation and checking as well as handling of acknowledgements and retransmission in case of errors. Figure 22 indicates the responsibilities of the application program and the EIB/KNX library with reference to the OSI 7-layer communication model.

Due to limited available resources of the hardware platform, the current implementation of the EIB/KNX library allows only the transmission and reception of one message at a time. If, for example, a message arrives and the last received message has not been processed, it will simply be discarded.

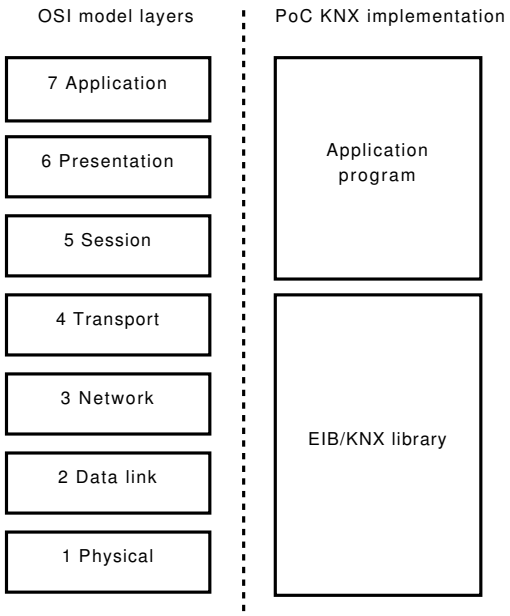


Figure 22: OSI model layer assignment between the application program and the EIB/KNX library.

6.2.3 Configuration

Especially when group objects are used, it is necessary to have a way for defining the EIB/KNX node configuration including the node's individual address, group objects, group addresses and the associations between them:

Individual address: The zone, line and device number of a device.

Group objects define the data points a node contains. A data type and an initial value needs to be specified.

Group addresses define the addresses which a node reacts to. The address consisting of the main group, optional mid group and sub group needs to be specified along with the permissions of this group address (read, write or both).

Associations are used to link group addresses to group objects. E.g. if a write message is received for a certain group address with the adequate permissions, the value is written to all associated group objects.

This information is stored in the internal EEPROM of the ATMega168 and loaded to SRAM at runtime. Figure 23 shows the memory layout of the configuration. A, B and C indicate the number of present entities in the configuration.

In the typical test setup only $A = B = C = 8$ was used (i.e. 8 group objects, 8 group addresses and 8 associations) as it was more than sufficient. However, the number of entities can easily be changed in the sources of the NanoVM and the configuration converter. Due to the binary representation of the associations, a maximum of $2^4 - 1 = 15$ (one pattern is used to indicate an unused association) group objects and group addresses is supported. The number of associations is only limited by the target memory.

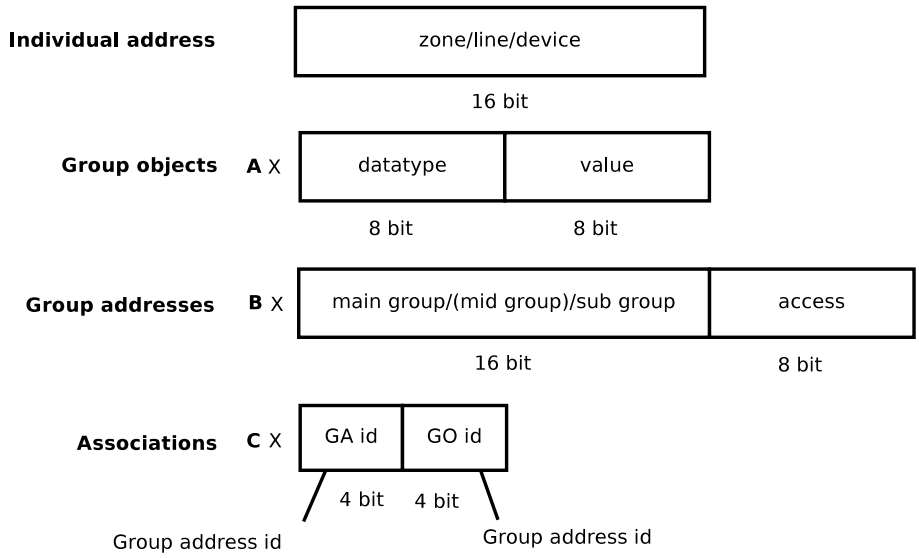


Figure 23: Memory layout of the configuration.

The configuration can be defined in an XML file which is used by a conversion tool to generate a binary representation of it which can be downloaded to the controller. The input format as well as the conversion tool is described in Section 6.4.

6.2.4 Invocation monitoring

As an attempt to make the execution of programs more controllable, an invocation monitor was developed as an extension of the NanoVM. It monitors the number of invocations of Java library methods and compares the recorded numbers to a set of rules defined at compile time. These rules can be used to specify the maximum number of invocations of a method during a defined timespan. If a rule is hurt, several actions may be taken, from simply ignoring the method call to halting the program. It would also be conceivable to have some kind of reporting to centralized entities in future versions of the monitoring implementation.

In the present version, the invocation rules have to be defined as a C array in a source file of the VM (`sec_policy.c`). The rules have to be defined as elements of an array of type `policy_rule_t`. A rule consists of a class reference, a method reference, an action, invocation limit and time limit. Each rule basically specifies how often a method may be invoked in the given time limit. Listing 2 shows an example for such a set of rules. Here, the method `txSend` may for example only be called at most 5 times in a timespan of 1 second. The listing also shows the use of the helper macro `COMPOSE_RULE` which simplifies the rule definition as the parameters are tightly packed. It has the following format:

`COMPOSE_RULE(invocation limit, time limit, action)`

where the time limit has to specified in tenths of a second and the action may currently be one of:

- RULE_ACTION_IGNORE
- RULE_ACTION_HALT
- RULE_ACTION_RESET

The rule checking was implemented to be as efficient as possible but due to the limited processing power of the underlying hardware it still causes a significant slowdown. This is primarily caused by the fact that every rule needs to be checked on each library method invocation. Additionally, a timing mechanism is required to keep track of the time spans between the invocations. The timing resolution was chosen at tenths of a second as a compromise between performance loss and flexibility. In general, the number of rules should be kept as low as possible in order not to slow down the execution of the VM too much.

Listing 2: Example of a small set of invocation rules.

```
policy_rule_t sebas_sec_policy_rules[SEBAS_SEC_POLICY_RULES_CNT]  =
{
  {NATIVE_CLASS_KNX, NATIVE_METHOD_TX_SEND, COMPOSE_RULE(5, 100,
       RULE_ACTION_HALT)},
  {NATIVE_CLASS_KNX, NATIVE_METHOD_GO_SET_VALUE, COMPOSE_RULE(3, 100,
       RULE_ACTION_HALT)},
};
```

6.3 Programming Framework

In this section, the programming framework will be described. First, a few examples of simple programs will be described to demonstrate the use of the programming libraries. Then, a reference for both the EIB/KNX as well as the seBAS I/O library will be given.

6.3.1 Examples

First, a simple example realizing a blinking LED will be presented. Afterwards, the group object and low-level interfacing mode will be compared by presenting two examples implementing the same functionality by using one of the interfaces respectively. The last example will demonstrate the combined use of the low-level and the high-level interface by using the fallback option.

Blinking LED

This example is intended to demonstrate the simplicity of programming user applications for the PoC. In general, the code in NanoVM programs is only related to the intended functionality and does not require any system specific initialization or management code. In Listing 3 it can be seen that only the main method is defined. First, a descriptive string is printed using `System.out`, which in the NanoVM standard I/O is mapped to the USART. This way, if a PC or other terminal device is connected, the successful loading of the correct program can be checked. Then in an infinite loop, the LED is continuously toggled, with a pause of half a second between each turning.

Listing 3: Blinking LED example.

```
package examples.sebas;

import nanovm.sebas.seBAS;

class LED {

  public static void main(String[] args) {
    System.out.println("seBAS LED example");

    while(true) {
      seBAS.greenLED(seBAS.ON);
      seBAS.wait(50);
      seBAS.greenLED(seBAS.OFF);
      seBAS.wait(50);
    }
  }
}
```

Simple Bus Interaction 1 - Group objects

The first version of the simple bus interaction in Listing 4 uses group objects for communicating with the EIB/KNX bus system. For this example, a node configuration defining two group objects (0 and 1) is assumed (like in Listing 9). Both with a binary data type, group object 0 is used as a sink object controlling the LED and group object 1 as source object switched by the pushbutton. How

the group objects are associated with group addresses can be arbitrarily defined in the configuration.

Again, first a descriptive string is printed. Then the interface mode is set to group objects only, which is actually unnecessary since it is the default mode. The main application code is again placed in an infinite loop. Here, first group object 0 is checked for changes which could be present if an related EIB/KNX message had been received. If there are any, the value is read and simply used to set the LED accordingly. Next it is checked if the pushbutton has been pressed. If yes, the boolean variable `toggle` is inverted and its value is written to group object 1. This value is then automatically sent to all associated group addresses.

Listing 4: Simple bus interaction using the group objects interface.

```
package examples.sebas;

import nanovm.sebas.seBAS;
import nanovm.knx.KNX;

class GroupObjects {

  public static void main(String[] args) {
    System.out.println("seBAS group objects example");

    boolean toggle = false;

    // Set interface mode to group objects
    KNX.setInterfaceMode(KNX.GO_ONLY);

    while(true) {
      if ( KNX.goChanged(0) ) {
        seBAS.greenLED( KNX.goReadValue(0) );
      }

      if ( seBAS.buttonPressed() ) {
        toggle = !toggle;
        if (toggle)
          KNX.goWriteValue(1, 1);
        else
          KNX.goWriteValue(1, 0);
      }
    }
  }
}
```

Simple Bus Interaction 2 - Low level interface

The second version of the simple bus interaction in Listing 5 uses the low-level interface of the KNX library. After the interface mode is set to low-level only, the outgoing message is prepared. It is reset, the receiver is specified and the data length is set to 2. Then, again in an infinite loop, on the one hand the library is checked for received messages and on the other hand the state of the button is monitored. If a message with a data length of 2 was received (standard KNX

commands for binary values always consist of 2 bytes) the LED is set according to the received command. Afterwards the received message is discarded, which is an important operation. Because as long as a received message has not been discarded, all incoming messages are ignored. In case the button was pressed, the message for transmission with the alternating value of the `toggle` variable as data is completed and sent.

Listing 5: Simple bus interaction using the low-level interface.

```
package examples.sebas;

import nanovm.sebas.seBAS;
import nanovm.knx.KNX;

class LowLevel {

  public static void main(String[] args) {
    System.out.println("seBAS low-level interface example");

    boolean toggle = false;

    // Set interface mode to group objects
    KNX.setInterfaceMode(KNX.LL_ONLY);

    // prepare message
    KNX.txReset();
    KNX.txSetReceiver((short)0x1155);
    KNX.txSetDataLength(2);

    while(true) {
      // check if a message was received
      if (KNX.rxReceived()) {
        if (KNX.rxGetDataLength() == 2) {
          short s = KNX.rxGetByte(1);

          if (s == 0x81)
            seBAS.greenLED(seBAS.ON);
          else if (s == 0x80)
            seBAS.greenLED(seBAS.OFF);
        }
        KNX.rxDiscard();
      }

      if (seBAS.buttonPressed()) {
        toggle = !toggle;

        if (toggle)
           KNX.txSetByte(1, 0x81);
        else
          KNX.txSetByte(1, 0x80);

        KNX.txSend();
      }
    }
  }
}
```

Simple Bus Interaction 3 - Combined interfaces(fallback mode)

The combined version of the simple bus interaction example is depicted in Listing 6. The interface mode is set to low-level fallback. After that the example is similar to the one solely using the group objects interface. The only difference is that after polling the group object value for changes, it is also checked if the low-level interface signalizes the reception of a message. This will only be the case if a message was received which could not be processed by the high-level interface. If signalled, the message can then be processed with the low-level interface (only indicated in the example).

Listing 6: Simple bus interaction using both the high-level and low-level interfaces in fallback mode.

```
package examples.sebas;

import nanovm.sebas.seBAS;
import nanovm.knx.KNX;

class Fallback {

  public static void processCustomMessage() {
    // if (KNX.rxGetByte(0) == ...
    // Implementation omitted
  }

  public static void main(String[] args) {
    System.out.println("seBAS fallback example");

    boolean toggle = false;

    // Set interface mode to group objects
    KNX.setInterfaceMode(KNX.LL_FALLBACK);

    while(true) {
      if ( KNX.goChanged(0) ) {
        seBAS.greenLED( KNX.goReadValue(0) );
      }

      // check if a message could not be processed
      // by the group object interface
      if (KNX.rxReceived()) {
        // process received message bytewise
        processCustomMessage();

        // Important, otherwise both low-level and high-level
        // interface are blocked
        KNX.rxDiscard();
      }

      if ( seBAS.buttonPressed() ) {
        toggle = !toggle;
        if (toggle)
          KNX.goWriteValue(1, 1);
        else
          KNX.goWriteValue(1, 0);
      }
```

```
    }
  }
}
```

6.3.2 KNX library reference

Variables

```
int GO_ONLY = 1;
```

```
int LL_FALLBACK = 2;
```

```
int LL_ONLY = 3;
```

General

`void setInterfaceMode(int mode);`
: Can be used for defining the desired messaging interface mode (GO_ONLY, LL_FALLBACK or LL_ONLY). Default is GO_ONLY.

Group object interface

`boolean goChanged(int index);`
: Returns the state of the group object with the specified index. Returns true if the group objects was changed since the last call to `goChanged`.

`short goReadValue(int index);`
: Reads and returns the current value of the group object with the specified index. The current value in memory is read, the operation does not cause any messages to be sent.

`void goWriteValue(int index, int value);`
: Writes the value of the group object with the specified index. If group addresses with write access are associated with the group object, the value is sent to the bus.

`void goRequestUpdate(int index);`
: Requests updating the group object with the specified index. Causes sending of a read message for each associated group address with write access.

Low-level interface

`void txSetByte(int index, int data);`
Sets the byte with the specified index in the outgoing message data buffer.

`void txSetDataLength(int len);`
Sets the data length of the data buffer of the outgoing message.

`void txAddByte(int data);`
Adds a byte to the data buffer of the outgoing message. Writes a byte to the current end of the data buffer and increases the data length by 1.

`void txSetReceiver(short receiver);`
Sets the receiver for the outgoing message.

`void txSetPriority(short receiver);`
Sets the priority for the outgoing message.

`void txSend();`
Blocking send of the outgoing message.

`void txReset();`
Resets the outgoing message.

`short rxGetByte(int index);`
Returns the byte with the specified index of the incoming message data buffer.

`byte rxGetDataLength();`
Returns the length of the incoming message data.

`short rxGetSender();`
Returns the sender address of the incoming message.

`byte rxGetPriority();`
Returns the priority of the incoming message.

`boolean rxReceived();`
Polls for received messages waiting to be processed.

`void rxDiscard();`
Discards the current stored received message. Should be invoked as soon as a received message has been processed. As long as a message is in the receive buffer and has not been discarded, all incoming messages are ignored (but counted, see `rxMissedCount()`).

`byte rxMissedCount();`
Returns the number of missed messages due to a blocked incoming message buffer.

Configuration related

`short nodeIndividualAddress();`
Returns the configured individual address of the node.

Debugging

`void signalBus();`
Can be used to signal a bit on the bus (solely for debugging purposes).

6.3.3 seBAS library reference

Variables

```
int OFF = 0;

int ON = 1;
```

Peripherals

`void greenLED(int state);`
Controls the green led on the PoC board. The `state` parameter can be either ON or OFF.

`boolean buttonPressed();`
Checks if the button has been pressed since the last call to this function.

Other

`void wait(int hsec);`
Waits the specified time (in hundreths of a second).

6.3.4 File structure

The file structure of the Proof-of-Concept software solution is outlined in Listing 7. In the listing, only the PoC-relevant directories are shown. The base directory has only two subdirectories, `nanovm` and `hardware`. Since the PoC software is basically a modified version of the NanoVM, the file structure in the `nanovm`

directory is for the most part the same as in the current version of the NanoVM code distribution. Only a few additions are present, like the configuration directory. The `hardware` folder currently only holds the schematics of the assembled hardware platform, along with part lists.

Listing 7: Directory structure of the PoC software.

```
../poc/
|-- hardware
|   `-- schematics
`-- nanovm
    |-- config
    |   |-- desc
    |   |-- examples
    |   |   `-- simple
    |   `-- src
    |-- doc
    |-- java
    |   |-- examples
    |   |   `-- sebas
    |   `-- nanovm
    |       |-- io
    |       |-- knx
    |       |-- sebas
    |       `-- util
    |-- mybin
    |-- tool
    |   |-- config
    |   `-- src
    `-- vm
        |-- build
        |   |-- avr_mega168
        |   |-- sebas
        |   `-- unix
        `-- src
            `-- sebas
```

In the following, the relevant subdirectories of the `nanovm` directory are described:

`config` holds configuration related files. `desc` contains the XML schema for configuration definitions in XSD format. `src` holds the conversion script for generating binary representations of a configuration. Example configurations are situated in `examples`.

`doc` includes the NanoVM documentation.

`java` holds all Java resources of the PoC. `nanovm` contains the Java packages of the native library. In `sebas` the PoC I/O library and in `knx` the EIB/KNX library is located.

`mybin` contains some helper scripts for uploading application programs to the PoC.

tool holds the NanoVMTool. `config` contains the configuration including the mapping of Java methods to numerical identifiers. The tool source is situated in `src`.

vm holds the VM implementation. The C sources are located in in `src`, the subdirectory `sebas` contains PoC related code. The relevant files are:

- `native.h` defines the method identifiers.
- `native_impl.c` NanoVM integration file for the native libraries.
- `native_knx.(c,h)` holds the EIB/KNX library implementation.
- `native_sebas.(c,h)` holds the PoC I/O library implementation.
- `exteep.(c,h)` for accessing the external EEPROM.
- `sec.(c,h)` for the invocation monitoring.
- `sec_policy.(c,h)` defines the invocation monitoring rules.

The `build` directory holds makefiles for building the VM.

6.4 Tools

A number of standard tools are necessary used for compiling, preparing and running the PoC. For compilation of the VM, avr-gcc 4.2.2 with its provided tools (avr-objdump, avr-objcopy) and GNU Make 3.81 have to be used. The Java programs are compiled using javac 1.6.0_06. Binary files are loaded into the microcontroller's memory using the AVR Downloader/UploaDEr (avrdude) 5.5-1.

Additionally, the NanoVMTool, as provided by the NanoVM distribution, is required for preparing and uploading Java application programs. For converting the EIB/KNX node configuration to a binary format, a conversion tool was developed, its working title is currently seBASConf. These tools shall be described in greater detail in this section.

Most tasks are automated, either using flexible makefiles or small scripts. For example, compiling and uploading the VM is simply done by executing `make install` in the working directory. The same is valid for the configuration. For uploading application programs, a small script is provided in `nanovm/mybin/inst` which simply takes the class name to be uploaded as a parameter. This requires that the corresponding Java file is placed in the `nanovm/java/examples` directory.

NanoVMTool

Since the NanoVM cannot handle standard Java class files, the NanoVMTool is required. It combines the program code of all used classes into a so-called NVM file. Additionally, it strips all unnecessary data from class files to save memory on the target system. The NanoVMTool can then upload the file to the directly to a running NanoVM for which it uses the RXTX libraries (www.rxtx.org). File transmission on the side of the NanoVM is handled by the built-in bootloader which listens for incoming transmissions at system start.

The NanoVMTool is distributed as translated JAR binary ready for use. It has the following synopsis:

```
java -jar NanoVMTool.jar <system config> <root dir>
<class name>
```

The first parameter is the location of a config file which is required for defining various settings like maximum code size, the device for uploading the program along with the baud rate and files containing native mappings to be included. These mappings are used for relating Java library methods to numerical identifiers which are used by the C implementation. The second parameter sets the base directory for the Java class to be uploaded and the third parameter names the class.

seBASConf - Configuration

seBASConf is a script written for parsing an EIB/KNX configuration in XML format and converting it to Intel hex format. It was written in Python which provides powerful built-in XML handling mechanisms. The required XML schema for the configuration is defined in an XSD template which is provided with the tool. The schema is for validating the configuration by the user. In the current release, it is not automatically validated by the tool. This functionality may be added in future versions. However, invalid XML will nevertheless lead to errors during parsing.

The synopsis for the tool (given execution permissions of the script) is the following:

```
seBASConf.py <XML Configuration> <output size>
<output file>
```

The first parameter has to be the path of the EIB/KNX configuration description. The second parameter is the desired output size which depends on the memory which will be used for storing the configuration. In the PoC it is the internal EEPROM of the ATMega168 which is 512 bytes large. The output is padded with zeros to fill up the specified size as the tool used for uploading requires the .hex file to be the same size as the target memory. The last parameter is the filename

of the destination .hex file.

Once the configuration has been processed, the resulting .hex file can directly be uploaded to the target system. During the development of the PoC, `avrdude` was used for this task.

The configuration consists of an individual address, group objects, group addresses and associations between the latter two. The parameters have already been described in Section 6.2.3. The XML schema is quite straightforward, its current version is shown in Listing 8.

Listing 8: XML schema for the configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.auto.tuwien.ac.at/seBASConfigurationSchema"
     xmlns="http://www.w3.org/2001/XMLSchema" xmlns:sebasconf="http://www.auto.
     tuwien.ac.at/seBASConfigurationSchema">

    <complexType name="seBASConfigurationType">
      <sequence>
        <element name="individualAddress" type="sebasconf:IndividualAddressType"
             ></element>
        <element name="groupObjects" type="sebasconf:GroupObjectSequenceType"></
             element>
        <element name="groupAddresses" type="sebasconf:GroupAddressSequenceType"
             ></element>
        <element name="associations" type="sebasconf:AssociationSequenceType"></
             element>
      </sequence>
    </complexType>

    <complexType name="GroupObjectSequenceType">
      <sequence>
        <element name="groupObject" type="sebasconf:GroupObjectType" minOccurs="0
             " maxOccurs="8"></element>
      </sequence>
    </complexType>

    <complexType name="GroupAddressSequenceType">
      <sequence>
        <element name="groupAddress" type="sebasconf:GroupAddressType" minOccurs=
             "0" maxOccurs="8"></element>
      </sequence>
    </complexType>

    <complexType name="AssociationSequenceType">
      <sequence>
        <element name="association" type="sebasconf:AssociationType" minOccurs="0
             " maxOccurs="8"></element>
      </sequence>
    </complexType>

    <complexType name="GroupObjectType">
      <attribute name="id" type="integer" use="required"></attribute>
      <attribute name="dataType" type="string" use="required"></attribute>
      <attribute name="value" type="integer" use="required"></attribute>
    </complexType>

    <complexType name="GroupAddressType">
      <attribute name="id" type="integer" use="required"></attribute>
```

```
      <attribute name="mainGroup" type="short" use="required"></attribute>
      <attribute name="midGroup" type="short"></attribute>
      <attribute name="subGroup" type="short" use="required"></attribute>
      <attribute name="access" type="string" use="required"></attribute>
    </complexType>

    <complexType name="AssociationType">
      <attribute name="groupObjectId" type="integer" use="required"></attribute>
      <attribute name="groupAddressId" type="integer" use="required"></attribute>
    </complexType>

    <complexType name="IndividualAddressType">
      <attribute name="zone" type="short" use="required"></attribute>
      <attribute name="line" type="short" use="required"></attribute>
      <attribute name="device" type="short" use="required"></attribute>
    </complexType>

    <element name="seBASConfiguration" type="sebasconf:seBASConfigurationType"></
         element>

</schema>
```

Listing 9 shows an example for a very simple configuration file to be used with seBASConf. In this example two group objects (0 and 1) are defined and each is associated with a group address, one with writing and one with reading access.

Listing 9: A simple configuration example.

```
<?xml version="1.0" encoding="UTF-8"?>
<sebasconf:seBASConfiguration xmlns:sebasconf="http://www.auto.tuwien.ac.at/
     seBASConfigurationSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
     instance" xsi:schemaLocation="http://www.auto.tuwien.ac.at/
     seBASConfigurationSchema seBASConfigurationSchema.xsd ">

    <individualAddress zone="5" line="5" device="85"/>

    <groupObjects>
        <groupObject id="0" dataType="DTP_BOOL" value="1"/>
        <groupObject id="1" dataType="DTP_BOOL" value="0"/>
    </groupObjects>

    <groupAddresses>
        <groupAddress id="0" mainGroup="0" midGroup="0" subGroup="3"  access="r"
             />
        <groupAddress id="1" mainGroup="0" midGroup="0" subGroup="1" access="w"/>
    </groupAddresses>

    <associations>
        <association groupObjectId="0" groupAddressId="0"/>
        <association groupObjectId="1" groupAddressId="1"/>
    </associations>

</sebasconf:seBASConfiguration>
```

7 Experiences/Results

The Proof-of-Concept was shown to offer decent flexibility which enables its use in many applications. In its current state it could be used to carry out typical functionality of a standard EIB/KNX bus coupling unit (BCU). With its attached peripherals it could already be used to implement a number of simple applications. The standard example is a toggling light switch which was already tested during development. More complex switching tasks with several binary in- and outputs would easily be possible.

If external equipment would be attached to the PoC, which can be done via the provided Freebus interface, the possible uses are even more diversified. For example, with a temperature sensor attached it could be setup as a heating controller. With more pushbuttons attached it could control dimmed lights or sun-blinds.

7.1 Memory consumption

In order to check how much resources the implemented Proof-of-Concept requires, some investigations were done regarding its memory consumption. The NanoVM uses a virtual heap architecture to provide the necessary memory for the Java programs. Because it is difficult to measure how much memory was really used on such a system, the data provided by the heap managing part of the NanoVM was evaluated. In its original version, the heap size the NanoVM provides on an ATMega168 is 768 bytes, leaving 256 bytes for the operation of the NanoVM itself, which should be enough under all circumstances (according to the author of the NanoVM). With the extensions that have been implemented in the Proof-of-Concept, approximately 70 bytes of extra initialized memory are used. To be on the save side, the heap size was reduced by 100 bytes, to 668 bytes.

The following experiment was used to get an approximation of the actual memory consumption of a typical Java program. At first a number of actions, involving switching of the LED, EIB/KNX reception and transmission, were executed and then the remaining heap size was inspected. This was repeated several times. As expected, the heap size was shrinking every cycle until it reached zero. Then the built-in garbage collector was run, freeing all the unused heap memory. The heap size which is freed in this step is considered to be the actual unused heap memory which is available to applications. In the experiment with the fully implemented Proof-of-Concept, this size was around 580 bytes, which makes a program memory use of around 90 bytes. This indicates that there is quite a lot of room left for complex programs.

Matrix dimension	Native C	Java (NanoVM) int. EEPROM	Java (NanoVM) ext. EEPROM
2	31 μ s	13.2 ms	216 ms
4	440 μ s	20ms	1.34 s
8	3.92 ms	595 ms	9.60 s
16	32.80 ms	4.64 s	-

Table 2: Measured durations of a square matrix multiplication with different matrix dimensions using different implementations.

7.2 Performance

The performance of the Proof-of-Concept implementation was evaluated from several points of view. When it comes to raw calculation capacity, the implementation is, as expected, very slow. Table 2 shows some performance measurements of a square matrix multiplication with different matrix dimensions[1]. Implementations in C and Java, running in the NanoVM using either internal or external EEPROM as program storage, are compared.

The measured durations of the native C implementation are always three to four orders of a magnitude smaller than the corresponding Java versions. Memory access operations have been identified as a major reason for slowdown in NanoVM programs. But even if such operations are removed, basically making the operation only correspond in the number of multiplications and additions, the overall operation durations are nowhere near the native C implementations.

It can be argued though, that the purpose of the system has never been the maximization of performance but rather security and, to some extent, flexibility. Therefore, it is acceptable that the performance is poor, as long as it is good enough to enable the system to fulfill the required tasks. In the special case of our Proof-of-Concept implementation, these tasks are control functions, which are usually not very computationally expensive. For example consider a heat controller, one of the more complex devices in a building automation system. If such a device is realized as a PID controller, every controller cycle requires only a few additions and multiplications.

The performance is also good enough to enable a decent responsiveness in applications with user interaction, for example when the tapping of a pushbutton triggers certain operations. It also has to be noted that the current clock of 8 MHz is really on the lower end of the considered target systems. Even the current microcontroller could be clocked at up to 20 MHz which would probably double

[1]The 16x16 matrix multiplication was not measurable with the used oscilloscope when the external EEPROM was used as program storage. The maximum measurable duration was 50 seconds. The multiplication takes roughly 75 seconds.

the performance.

Finally, it is of course possible to implement performance critical functionality in C and make it available to the user application through API methods. This would reduce the overhead imposed by the Java interpretation significantly.

7.3 Stability

During all tests carried out during the development of the PoC, its stability was generally satisfactory. Several times, a Java program was running actively for several days without any noticed errors. The transmission of messages over the EIB/KNX bus was very stable, no message losses could be detected.

Some stability problems related to the reception of messages were observed during testing. Only about 5 of 6 messages are received without errors while one usually contains a few bit errors all roughly at the same part of the message. The issue was analyzed by making visible the times where the value of a bit is taken from the bus. Before storing the value, the LED on the PoC board was lit and afterwards it was turned off. While the timespan is too short for the human eye, it was made visible using an oscilloscope.

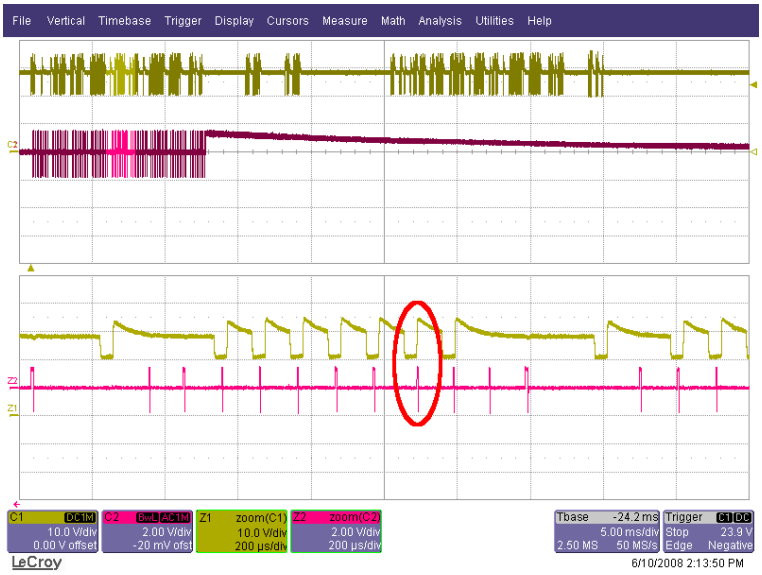


Figure 24: Analyzing the stability problems in the reception of messages.

Figure 24 shows such a visualization of a received erroneous message, in this example the message had one wrong bit. In the figure, the time of the storage of this bit is highlighted. The problem is obvious: due to some reason, the storage

is delayed until slightly after the positive flank of this bit. The exact voltage level on the bus at this time cannot be determined, but apparently it was already high enough to be detected as a logical one.

The reason for the delay was traced back to the heavy utilization of the microcontroller. The detection of a starting data frames is implemented with an external interrupt triggered by a falling edge. At the same time several other non-interruptible interrupt service routines (ISR) may be in use. For example, the NanoVM reads its instruction from the external EEPROM attached via I^2C bus which requires regular calls of an ISR. The invocation monitoring also uses a timer ISR. If a falling edge or a timer overrun is detected while another ISR is currently being processed, for example when the next instruction is read from the external EEPROM, it will not be handled until the routine is finished.

Since the system is only a PoC, not too much effort was used to solve this issue, although several ideas have been tried out without much success. A more efficient implementation or more clever use of timers may improve the situation. The problem may be subject to future improvements.

7.4 Freebus basic circuit

A side goal the Proof-of-Concept development was to evaluate the usefulness of the Freebus basic circuit in an environment of retail devices. In the PoC test environment, some reasonable comparisons were made and interoperability could be tested. This actually only concerns the signal levels produced since the EIB/KNX stack was developed from ground up. Although the basic circuit generally works as promised, early tests showed a slight problem. An EIB/KNX bus line holds a voltage of almost 30 V when idle. This state corresponds to a logical one. To signal a logical zero, a device may pull down the bus voltage to a minimum of 19 V, according to the EIB/KNX specification. The problem observed in the tests occurs when the basic circuit is used to signal a zero. In this case the voltage is pulled down to 16 V which is too low.

Figure 25 shows some measurements taken with the original basic circuit and with some modifications. A voltage characteristic as produced by a retail device (in this case a pushbutton by Siemens) is shown in yellow. The red characteristic is produced by the PoC board. The difference in the minimum level is obvious (there are also slight timing differences which can be neglected in these measurements). As a first approach to fix this issue, it was tried to limit the voltage drop by using a series resistor at the base of the 2N7000 MOS-FET. The blue characteristic was recorded with a 15 kOhm series resistor and green with 20 kOhm. It can be seen that with the series resistors, the voltage drops only to the desired levels but at cost of the steepness of the flanks.

Better results were obtained without series resistors by just lowering the oper-

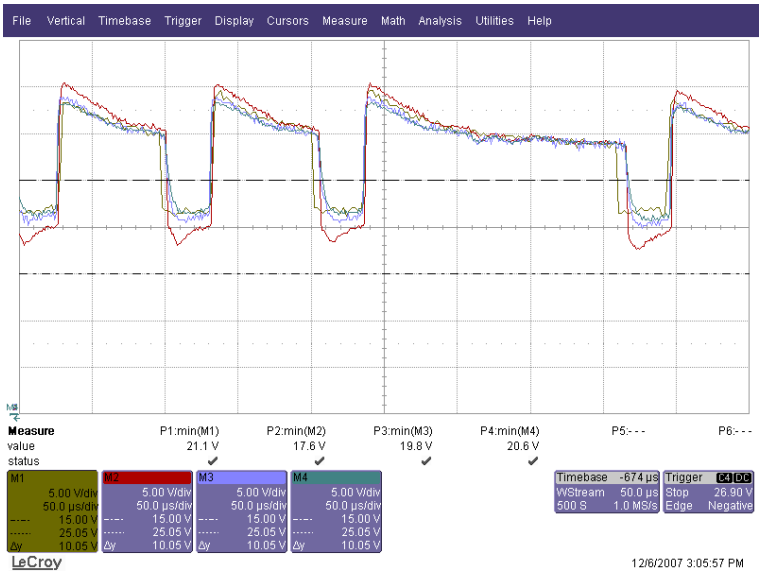


Figure 25: Voltage characteristics comparison.

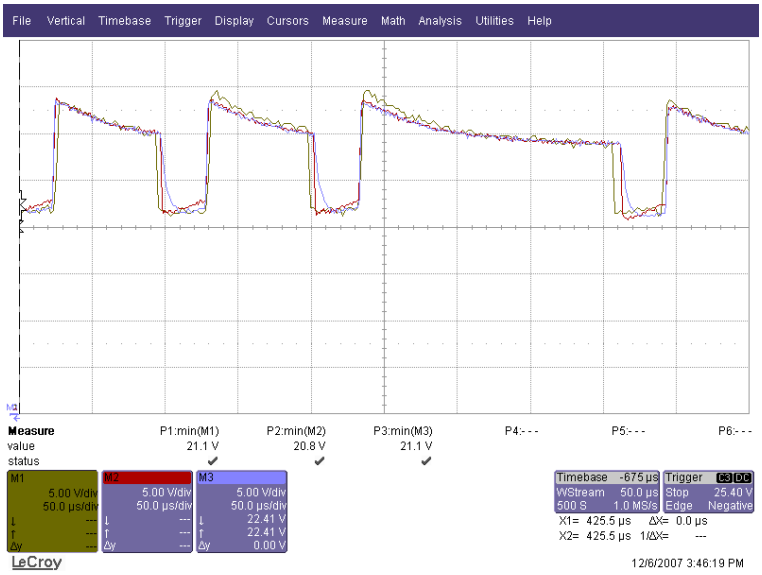


Figure 26: Voltage characteristics comparison.

ation voltage of the microcontroller. In Figure 26, measurements taken with such a setup are shown. The yellow curve again shows the characteristic of a retail device. Here, the red characteristic is taken with an operation voltage of 3.1 V, the blue one as comparison with a series resistor of 20 kOhm and 3.2 V operation voltage. The red characteristic looks very much like an original one, with decent steepness of its flanks.

However it has to be noted, that the circuit without any modifications was used for the major time of experimenting with the PoC. And although the signal characteristics fall slightly out of the specifications, no problems with the transmission of signals could be observed.

8 Summary and Outlook

The proposed architecture as introduced in this thesis represents a promising approach for providing application security in low-end embedded systems. It is a reasonable combination of adapted security mechanisms which allows fine-grained control over the execution of a user application. At the same time, a decent flexibility is provided to enable its utilization for many typical building automation tasks. Furthermore, other benefits like program portability and programming simplicity can be achieved. The architecture has already been published at the 13th IEEE International Conference on Emerging Technologies and Factory Automation [57].

A Proof-of-Concept was implemented to demonstrate the possibilities which can be enabled with the described secure architecture. It has a significant potential which was shown in a number of tests and experiments. The PoC is, naturally, not a complete solution ready to be used. Therefore, there is of course a number of improvements which are thinkable. Some of the ideas which emerged during the development are described here:

- The implementation of the system software could be optimized to achieve a more efficient microcontroller utilization. So far, not much effort was used for this task. It may increase the performance and improve the stability of the networking functions (especially message reception).

- Currently, the rules for the invocation monitoring have to be placed in a C file as part of an array. A more comfortable solution would be desirable. For example, an XML file could be used for the definition of the rules and a conversion utility could be used as part of the building process of the VM.

- The EIB/KNX node configuration, after it has been converted into binary form, is currently uploaded to the ATMega168's internal EEPROM by using the controller's In-System-Programming interface. This means that full access to the all memory regions is given. However, the system software situated in the flash memory should remain untouchable during the task of changing the configuration. Therefore, it would be optimal if the configuration could be uploaded through the NanoVM bootloader, just like the user application.

- Even though the Freebus basic circuit was shown to enable access to an EIB/KNX bus system, the use of a TP-UART for EIB/KNX interaction would have several advantages. The microcontroller would be relieved of the performance consuming low-level networking and the produced signals would be guaranteed to be standard compliant.

List of Figures

List of Tables

References

[1] *FreeBus website*. http://www.freebus.org/.

[2] A. S. Tanenbaum and M. van Steen, *Distributed Systems - Principles and Paradigms*. Prentice-Hall, 2002.

[3] C. P. Pfleeger and S. L. Pfleeger, *Security in Computing*. Prentice Hall Professional Technical Reference, 2002.

[4] *US-CERT Vulnerability Notes Database*. http://www.kb.cert.org/vuls.

[5] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted runtime monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 12, pp. 1295–1308, 2006.

[6] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *17th International Conference on VLSI Design, 2004. Proceedings.*, pp. 605–611, 2004.

[7] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, no. 6, pp. 644–654, 1976.

[8] R. L. Rivest, A. Shamir, and L. M. Adelman, "A method for obtaining digital signatures and public-key cryptosystems," Tech. Rep. MIT/LCS/TM-82, 1977.

[9] P. Zimmermann, *PGP source code and internals*. Cambridge, MA, USA: MIT Press, 1995.

[10] R. Rivest, "The md5 message-digest algorithm," 1992.

[11] U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Information Technology Laboratory (ITL)., *Secure Hash Standard*, August 2002. Federal Information Processing Standards Publication 180-2.

[12] H. Dobbertin, A. Bosselaers, and B. Preneel, "Ripemd-160: A strengthened version of ripemd," in *Fast Software Encryption*, pp. 71–82, 1996.

[13] *Common Criteria Portal*. http://www.commoncriteriaportal.org/.

[14] P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, July 2004.

[15] D. Hwang, P. Schaumont, K. Tiri, and I. Verbauwhede, "Securing embedded systems," *IEEE Security & Privacy Magazine*, vol. 4, no. 2, pp. 40–49, March-April 2006.

[16] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," *Lecture Notes in Computer Science*, vol. 1666, pp. 388–397, 1999.

[17] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *Transactions on Embedded Computing Systems*, vol. 3, no. 3, pp. 461–491, 2004.

[18] A. D. Rubin and D. E. Geer, Jr., "Mobile code security," *IEEE Internet Computing*, vol. 2, no. 6, pp. 30–34, 1998.

[19] W. Kastner, G. Neugschwandtner, S. Soucek, and H. Newmann, "Communication systems for building automation and control," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1178–1203, June 2005.

[20] W. Granzer, W. Kastner, G. Neugschwandtner, and F. Praus, "A modular architecture for building automation systems," in *IEEE International Workshop on Factory Communication Systems*, pp. 99–102, June 2006.

[21] *DALI Website*. http://www.dali-ag.org/.

[22] *M-Bus Website*. http://www.m-bus.com/.

[23] D. Loy, D. Dietrich, and H. Schweinzer, *Open Control Networks*. Kluwer Academic Publishers, 2002.

[24] *BACnet Website*. http://www.bacnet.org.

[25] *KNX Website*. http://www.knx.org/.

[26] W. Kastner and G. Neugschwandtner, "EIB: European installation bus," in *The Industrial Communication Technology Handbook* (R. Zurawski, ed.), vol. 1 of *The Industrial Information Technology Series*, ch. 34, pp. 34–1 — 34–18, Boca Raton: CRC Press, February 2005.

[27] D. Arora, S. Ravi, A. Raghunathan, and N. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Design, Automation and Test in Europe, 2005. Proceedings*, vol. 1, pp. 178–183, 2005.

[28] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[29] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2005.

[30] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, vol. 22nd Annual, pp. 269–278, December 2006.

[31] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 323–337, December 1992.

[32] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 2001.

[33] *ASTRÉE website*. http://www.astree.ens.fr/.

[34] *Sparse website*. http://www.kernel.org/pub/software/devel/sparse/.

[35] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," in *IEEE Transactions on Software Engineering*, vol. 28, pp. 735–746, August 2002.

[36] G. C. Necula and P. Lee, "Safe, untrusted agents using proof-carrying code," in *Mobile Agents and Security*, (London, UK), pp. 61–91, Springer-Verlag, 1998.

[37] L. Gu and J. A. Stankovic, "t-kernel: providing reliable os support to wireless sensor networks," in *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, (New York, NY, USA), pp. 1–14, ACM, 2006.

[38] J. McHugh, "Intrusion and intrusion detection," *International Journal of Information Security*, vol. 1, no. 1, pp. 14–35, 2001.

[39] Z. Li, A. Das, and J. Zhou, "Theoretical basis for intrusion detection," in *Proceedings of 6th IEEE Information Assurance Workshop (IAW)*, (West Point, NY, USA), IEEE SMC Society, June 2005.

[40] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[41] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications," in *Proceedings of the 6th Usenix Security Symposium*, (San Jose, CA, USA), 1996.

[42] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, 2005.

[43] M. Debbabi, M. Saleh, C. Talhi, and S. Zhioua, "Security evaluation of j2me cldc embedded java platform.," *Journal of Object Technology*, vol. 5, no. 2, pp. 125–154, 2006.

[44] P. Stanley-Marbell and L. Iftode, "Scylla: A smart virtual machine for mobile embedded systems," in *3rd IEEE Workshop on Mobile Computing Systems and Applications, WMCSA2000*, December 2000.

[45] *Java Platform Micro Edition website*. http://java.sun.com/javame.

[46] T. Harbaum, *The NanoVM - Java for the AVR*. http://www.harbaum.org/till/nanovm/index.shtml.

[47] J. H. Solorzano, *TinyVM- Java for LEGO Mindstorms*. http://tinyvm.sourceforge.net/.

[48] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *Digital Rights Management Workshop*, pp. 141–159, 2001.

[49] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, (London, UK), pp. 400–414, Springer-Verlag, 2003.

[50] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th USENIX Security Conference*, (San Antonio, Texas), pp. 63–78, jan 1998.

[51] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "Formatguard: Automatic protection from printf format string vulnerabilities," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2001.

[52] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proceedings of the 10th Network and Distributed System Security Sym posium*, (San Diego, California), pp. 149–162, February 2003.

[53] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pp. 483–488, 4-8 June 2007.

[54] I. Hiroaki, M. Edahiro, and J. Sakai, "Towards scalable and secure execution platform for embedded systems," in *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pp. 350–354, January 2007.

[55] R. Riley, X. Jiang, and D. Xu, "An architectural approach to preventing code injection attacks," in *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, (Washington, DC, USA), pp. 30–40, IEEE Computer Society, 2007.

[56] *AMD website*. http://www.amd.com.

[57] F. Praus, T. Flanitzer, and W. Kastner, "Secure and customizable software applications in embedded networks," in *13th IEEE International Workshop on Emerging Technologies and Factory Automation*, 2008.