Abstract

This Deliverable describes attacks on existing sensor nodes. We concentrate on two different kinds of attacks: (1) attacks on the cryptographic primitives of the nodes and (2) attacks on system level that exploit weaknesses on the implementation of the software of the node. The first group covers side-channel attacks, i.e. timing attacks, power measurements and analysis of electromagnetic emanations, while the second part discusses buffer overflow and stack overflow attacks in the context of sensor nodes.
Disclaimer
This document contains material, which is the copyright of certain TAMPRES consortium parties, and may not be reproduced or copied without permission.

In case of Public (PU):
All TAMPRES consortium parties have agreed to full publication of this document.

Impressum
TAMPREs - Tamper Resistant Sensor Node
TAMPRES
WP1 “Attack Analysis, Requirements and Methodology”
D1.2 - Analysis of attacks on sensor nodes software and hardware
[Editor: Name, company] Jörn-Marc Schmidt, IAIK, TU Graz

Copyright notice
©2011 Participants in project TAMPRES
## List of authors

<table>
<thead>
<tr>
<th>Company</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAIK</td>
<td>Jörn-Marc Schmidt</td>
</tr>
<tr>
<td>IAIK</td>
<td>Mario Kirschbaum</td>
</tr>
<tr>
<td>ETHZ</td>
<td>Aurélien Francillon</td>
</tr>
<tr>
<td>ETHZ</td>
<td>Manu Sekar</td>
</tr>
<tr>
<td>IHP</td>
<td>Frank Vater</td>
</tr>
<tr>
<td>IHP</td>
<td>Steffen Peter</td>
</tr>
</tbody>
</table>
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of authors</td>
<td>3</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>2 Targets</strong></td>
<td>6</td>
</tr>
<tr>
<td>2.1 Coalesenses iSense Core Module 2 Sensor Node</td>
<td>6</td>
</tr>
<tr>
<td>2.2 IHP Sensor Node</td>
<td>6</td>
</tr>
<tr>
<td><strong>3 Attacks on Cryptographic Primitives</strong></td>
<td>8</td>
</tr>
<tr>
<td>3.1 Attack Setups</td>
<td>8</td>
</tr>
<tr>
<td>3.1.1 Timing Attacks</td>
<td>8</td>
</tr>
<tr>
<td>3.1.2 Power Measurements</td>
<td>9</td>
</tr>
<tr>
<td>3.1.3 Electromagnetic Measurements</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Attacks on Coalesenses Node</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Attacks on IHP Sensor Node</td>
<td>11</td>
</tr>
<tr>
<td>3.3.1 Whitebox Attacks on AES Hardware module</td>
<td>11</td>
</tr>
<tr>
<td>3.3.2 Blackbox Attacks on IHP Sensor Node</td>
<td>14</td>
</tr>
<tr>
<td>3.3.3 Attacks on AES Software Implementation</td>
<td>15</td>
</tr>
<tr>
<td>3.3.4 Attacks on IHP Sensor Node - Hardware ECC</td>
<td>16</td>
</tr>
<tr>
<td><strong>4 Attacks on System Level</strong></td>
<td>19</td>
</tr>
<tr>
<td>4.1 Background: Software Attacks on General Purpose Computers</td>
<td>19</td>
</tr>
<tr>
<td>4.1.1 Code Injection Attacks</td>
<td>19</td>
</tr>
<tr>
<td>4.1.2 Malicious Code Execution Without Code Injection</td>
<td>21</td>
</tr>
<tr>
<td>4.1.3 Non Buffer Overflow-Based Software Attacks</td>
<td>22</td>
</tr>
<tr>
<td>4.2 Mitigation Techniques on General Purpose Computers</td>
<td>23</td>
</tr>
<tr>
<td>4.2.1 Preventive Measures</td>
<td>23</td>
</tr>
<tr>
<td>4.2.2 Protecting the Stack</td>
<td>23</td>
</tr>
<tr>
<td>4.2.3 Making Exploitation of Control-Flow Attacks Difficult</td>
<td>24</td>
</tr>
<tr>
<td>4.2.4 Protection by Modification of The Stack Model</td>
<td>25</td>
</tr>
<tr>
<td>4.2.5 Malicious Code Detection</td>
<td>26</td>
</tr>
<tr>
<td>4.3 Software attacks on WSN nodes</td>
<td>26</td>
</tr>
<tr>
<td>4.3.1 Generic Attacks on Microcontrollers</td>
<td>26</td>
</tr>
<tr>
<td>4.3.2 Hardware/software Attacks Specific to Microcontrollers and WSN Nodes</td>
<td>29</td>
</tr>
<tr>
<td>4.4 Evaluation of attacks on Coalesenses and IHP nodes</td>
<td>34</td>
</tr>
<tr>
<td>4.4.1 IHP Nodes</td>
<td>38</td>
</tr>
<tr>
<td><strong>5 Conclusion</strong></td>
<td>39</td>
</tr>
<tr>
<td><strong>A Validation of Software Attacks on BA2 Processor</strong></td>
<td>45</td>
</tr>
</tbody>
</table>
1 Introduction

This document describes attacks on sensor nodes. We point out that actual nodes are not sufficiently protected against adversaries that can gain physical access to the devices. Considering huge sensor networks that operate in environments with public access, this can pose a serious threat. This is because an adversary can steal a node, analyze it in laboratory conditions and maybe reveal sensitive information like a secret key to access the network. Unless the disappearing of a node is recognized in time and the keys are updated, the adversary can act as a node of the network and manipulate the whole system.

The first part of this document deals with attacks on the cryptographic hardware of the nodes. We show that using side-channel information, like the timing behavior, the power consumption and the electromagnetic emanation of the device, the secret key of a node can be determined. In particular, we present attacks on an off-the-shelf sensor node from Coalesenses as well as on a node developed by iHP. This first one comprises a hardware AES module, which was successfully attacked by a measurement of its electromagnetic emanations. The second device features an ECC hardware module, a hardware AES module, and a Leon2 processor core. We performed timing attacks on the ECC module of the iHP node and power analysis attacks on the hardware AES implementation as well as on an AES implemented in software running on the 32-bit Leon2 processor.

The second part of this document concentrates on the impact of attacks on software and memory on current sensor hardware. We first describe well known software attacks and describe their applicability to devices considered in this project. The attacks include code injection attacks as well as manipulation of the program flow by manipulating the memory unit and system stack. Major threats of the attacks are the modification of software and behavior of the nodes and software-based extraction of key material. We finally provide an analysis of the effectiveness of specific attacks on state-of-the art sensors including protocols of resources spent for the attacks.
2 Targets

For the analysis, we chose two different nodes. The first node is an off-the-shelf sensor node containing a Jennic microprocessor, while the second sensor node was developed by iHP comprising a Leon2 core and hardware co-processors for AES and ECC.

2.1 Coalesenses iSense Core Module 2 Sensor Node

The iSense Core Module 2 [19] provides the basis of the iSense modular hardware platform for all kinds of wireless sensor networking applications:

- iSense Core Module CM20 IEEE 802.15.4 compliant radio, 250kbit/s, hardware AES encryption, ToF ranging engine
- 32 Bit RISC Controller, 4-32MHz
- 128kB RAM, 512kB Flash
- Software controllable voltage regulator
- Expansion connectors
- Connector or integrated PCB antenna

iSense gives way to high performance and low power sensor networks.

Its radio interface is 802.15.4 compliant, Zigbee-ready radio is included, offering high data rates at ranges of up to 600m while providing hardware AES encryption. This IEEE 802.15.4 radio supports distance measurements to neighboring devices using time of flight ranging.

![Image of iSense Core Module 2](image)

Figure 1: iSense Core Module 2

The microcontroller core present in the iSense module is the Jennic [38], which itself is based on an Open-RISC 1200 [49] architecture provided by Beyond Semiconductors [57]. The Jennic chip has several interesting software security features such as stack overflow protection (as per 4.1.3). Hardware AES with an OTP memory for key storage.

2.2 IHP Sensor Node

As shown in Figure 2 the IHP sensor node is a complete sensor node system that provides several dedicated functions. On one hand there are two radio interfaces, so the system can interact as bridge between two different networks (802.15.4 and bluetooth). On the other hand the IHP sensor node offers special cryptographic accelerators. Like the iSense Core Module the IHP node has an AES hardware accelerator. Additionally the IHP sensor node has an accelerator for ECC and for SHA-1.

- 32 bit LEON2 CPU, typ. 16Mhz
Figure 2: The IHP Sensor Node

- 2 MB RAM
- 8 MB Flash
- Bluetooth Radio Interface
- 802.15.4 Radio Interface

The microcontroller of the node is the LEON2 [5] which is provided by Gaisler Research. The 32-Bit RISC-Processor is written in VHDL and is under LGPL available. It bases on SPARC-V8 architecture. The ASIC has separate data and instruction cache, interrupt controller, timer, debug interface and a simple power management. All components are connected by the AMBA [8] bus (see Figure 3). The processor core has a five stage instruction pipeline and a set of 8 register windows. Most of the instructions will be executed in one clock cycle.

A variety of operating systems have been ported for LEON2-based systems. The preferred operating system for the IHP sensor node is eCos [25]. It is a open source software for embedded systems and realtime applications. It configurable to adapt the system on special requirements for many application areas [9]. For the attacks we did not use the operating system because all crypto cores are very easy addressable on low level.
3 Attacks on Cryptographic Primitives

In this section, we discuss general attack types on the built-in cryptographic primitives of the sensor nodes. The attacks exploit physical properties of the devices like the timing behavior or the power consumption during operations that involve sensitive data.

3.1 Attack Setups

3.1.1 Timing Attacks

Needed time is one characteristic property of any sort of implemented algorithm. This property can be exploited for malicious side-channel attacks if the time needed for cryptographic operations reveals information on secret keys. For instance if the computation of a ciphertext needs characteristically more time for a key specific $k_1$ than for another key $k_0$ this is one information that can be exploited. Typically, the result of most timing attacks is not the key material directly, but valuable information, e.g. the number of '1's in the binary key, which help to reduce the search space significantly. This sort of attack is dangerous since generally the timing information is easily accessible.

Timing attacks can be executed on various levels. On a higher protocol levels [63] and [14] discussed remote attacks that measure the answering times of devices to obtain information of key material. These attacks may be a particular issue for software implementations of cryptographic primitives on sensor nodes since the low computation performance of the nodes increases the visibility of the timing behaviors.

Attacks, rather in focus of this work, measure the time for the execution of one cryptographic operation. Attacks for popular cryptographic systems such as Diffie-Hellman and RSA are described in [42]. In particular many implementations of RSA [53] are vulnerable against timing attacks, because typically RSA is implemented in a way that a '0' in the binary key causes a simple operation while a '1' needs significant computations.

On the lowest level the timing for sub-operations of the algorithm are measured. Therefore typically other side-channels, such as power measurements or operations on bus or memory, are exploited to identify the
individual sub-operations. For this sort of attacks already direct physical access to the devices and specialized equipment are needed to measure the time with sufficient precision. The advantage, however, is that this sort of attack may deliver the key directly.

3.1.2 Power Measurements

A conventional side-channel measurement setup is depicted in Figure 4. A host PC is used (1) to communicate with the device under test (DUT) over a serial, parallel, or USB connection, (2) to control a digital oscilloscope used for measuring the power consumption or the EM radiation of the DUT, and (3) to perform statistical analysis of the measured traces. The communication between the host PC and the oscilloscope is performed via GPIB, LAN, or over a direct connection (e.g. PCI) if the oscilloscope is integrated into the host PC. In our investigations we usually assume a strong attacker and hence we work with a trigger signal provided by the DUT which helps to align the measured traces. In Figure 4 also the opportunity of various fault injection methods is indicated.

![Typical power/em measurement setup.](image)

We perform our measurements with a LeCroy WavePro 725Zi digital oscilloscope [46]. The WP725Zi together with the active differential probe LeCroy AP034 [1] provides the required precision as well as the performance to investigate the side-channel resistance of state-of-the-art cryptographic devices.

We unite the operation of the basic tasks of a measurement setup like controlling an oscilloscope, performing cryptographic operations on the DUT, acquiring the measured traces, and statistical analyses of the traces by using the IAIK SCA toolkit which provides a huge set of MATLAB toolboxes.

3.1.3 Electromagnetic Measurements

The main advantage of EM measurements over power measurements is the flexibility regarding the investigation of specifically small areas on a chip die. Tiniest EM probes enable localized measurements of the emanation of single submodules within a digital circuit. Narrowing down the measured area and the involved modules in the circuit implicates a better quality of the measurements in terms of noise. This has a positive effect on the analysis of the measured traces.

For measurements of electromagnetic emissions, we use different coils (see Figure 7) that are mounted on a stepping device. The signals are measured using the oscilloscope described in Section 3.1 and analyzed using our Matlab toolbox. The toolbox is also capable of controlling the stepping device, a TANGO from Märzhäuser. It has an accuracy of \(+/−3\,\mu m\) and a resolution of \(0.01\,\mu m\). A picture of the setup is given in Figure 6.

3.2 Attacks on Coalesenses Node

We performed a black box correlation-based EM attack on the Jennic JN5148 wireless microcontroller. Black box attack means we do not exploit any detailed information of the sensor node (e.g. layout of the chip or internal structure of the microcontroller) during our attacks. The JN5148 module executes AES-128 encryptions / decryptions in counter mode: a nonce is concatenated with a counter value (increased after each encryption)
and is AES encrypted with a secret key. The result of the AES encryption $ciphertext'$ is XORed with the actual plaintext, which then results in the $ciphertext$. Figure 8 depicts the AES encryption in counter mode.

Figure 8: Positioning of the EM probe on the Jennic JN5148 wireless microcontroller unit under the shielding.

We performed a known plaintext / known ciphertext attack on the JN5148 module, i.e. we assume an attacker knows the plaintext and the ciphertext, and hence, he also knows the result of the actual AES encryption $ciphertext'$. We targeted the last round of the AES encryption and tried to reveal the last round key. We followed a straightforward approach and used the Hamming Weight of the output of the inverse S-Box operation in the last round of AES as hypothesis for our attacks.

In order to perform EM measurements on the JN5148 module we had to build a special measurement coil shown in Figure 7. The coil has been built using a 0.07 mm thin enameled copper wire and has an inner diameter of $\approx 0.25$ mm. The copper coil has been placed blindly between the Jennic IC and the shielding as shown in Figure 10. In a first measurement run we apparently placed the probe at a position on the microcontroller where no side-channel information from the AES security processor could have been measured. An example EM trace obtained from the first measurement is shown in Figure 11. The analysis of the first measurement did not reveal any useful information about the AES round key. We again placed the probe at different positions on the IC and inspected the measured EM emanation on the oscilloscope during the repetitive execution of AES operations. This way we discovered a probe position where we measured a slightly increased emanation during the AES operations. Figure 12 shows an example EM trace of the second measurement position: we
can see a slightly increased EM emanation in the middle area of the trace. At this position we were able to
unambiguously reveal 15 of 16 round key bytes\(^1\).

Figure 13 shows the correlation results of round key byte 6: we performed a correlation attack on 50 000
EM measurements, the correct key byte is plotted in red, the 255 incorrect key bytes are plotted in blue, the
correlation value is 0.0562. According to [47] the correlation peak $\rho$ in a successful attack can be used to
calculate the minimum number of required power traces to obtain unambiguous DPA results. For correlation
coefficients around the value 0.2 and below the following estimation holds: $n \approx \frac{28}{\rho^2}$ where the confidence
is 99.99%. According to that the number of required power traces $n$ for our correlation value of 0.0562 is
$n \approx 8800$. The correlation values of the other 14 successfully attacked round key bytes lie between 0.08 and
0.03, which corresponds to a number of required power traces between approximately 4000 and 30 000.

Figure 9: Specifically crafted copper coil for measuring the Jennic JN5148 module, inner diameter $\approx 0.25 \text{mm}$.

Figure 10: Positioning of the EM probe on the Jennic JN5148 wireless microcontroller unit under the shielding.

3.3 Attacks on IHP Sensor Node

3.3.1 Whitebox Attacks on AES Hardware module

The Figure 15 shows the measurement setup we used for attacking the device, which was previously analyzed.
The used ASIC provides an AES and an ECC core. The chip has a CardBus interface and it is the main
component of our CryptoCard. In contrast to the black-box attacks, we used a device containing only the crypto
cores itself for the whitebox approach. To measure the power consumption a shunt resistor was integrated in
the power supply.\(^2\)

The easiest possibility is the chosen plain text attack. In this case the attacker is able to choose the plain
text and to observe the power consumption. To identify which stored value results in which power consumption
we simulated a flip flop in an analogue way. As example see Figure 14. In this simulation we stored at first a
\(^1\)The 16\(^{th}\) round key byte could be easily revealed by a simple brute-force attack. A slightly different measurement position might
also reveal the last key byte.

\(^2\)For this attack, an oscilloscope Agilent 54854A measures the voltage drop over this 10 ohm resistor was used, rather than the
WP725Zi described in Section 3.1.2. The high resolution of 20GSamples/s helps to create an accurate mapping of the power consump-
tion in every clock cycle.
‘0’ and after that ‘1’. At 30ns and 40ns it is visible that the power consumption increases during storing a ‘1’ instead of a ‘0’.

The shown attack bases on the fact that only one bit of the input data will be changed. The correlation of the power consumption and the value of the bit (‘0’ or ‘1’) will be observed. The power will be integrated in the corresponding clock cycle.

Before every measurement the chip has to be reseted. The state machine gets their initial values and the register will be set on ‘0’. Now the ‘unknown’ key and the manipulated data will be load. For a bitwise recognition of the key, this bit of the data word will be manipulated, which will be added with unknown bit of the key. This AddKey operation is a bitwise XOR operation of the key and the corresponding data bit. The result will be stored in a register. The simulation of the power consumption of register (see Figure 14), helps to identify the stored value (whether ‘0’ or ‘1’).

The first step was to modify the pcb for the measuring. The capacitors has to be removed, otherwise they will be influence the measurement. The measurement was done at a frequency of 10 Megahertz. This frequency was evaluated during the setup phase. A lower frequency does not optimize the result, but a much higher frequency reduce the number of recognized key bits.

As described earlier the design has to be set in a known state before every measurement begins. This means the reset signal has to be set before starting. Due to the fact, that the initial phase of the chip and preparing the oscilloscope for a new measurement takes a lot of time, only 2 measurements per second can be performed.
Figure 14: Simulation of power consumption of a flipflop during change the stored value from ‘0’ to ‘1’

Figure 15: Measurements setup for DPA at the IHP
Every bit of the 128 bit key was observed separately. If the first bit of the key should be observed only the first bit of the data will be changed. In every measurement the corresponding data bit was flipped from ‘0’ to ‘1’ and reverse. From difference of the power consumption between these two measurements the value of the key bit can be estimated.

After an exemplary measurement of 600 measurements per bit approximately 121 of 128 bit of the key were correct recognized.

### 3.3.2 Blackbox Attacks on IHP Sensor Node

Similar to the JN5148 wireless module we performed black box correlation-based side-channel attacks on the AES coprocessor attached to the Leon2 processor core of the IHP sensor node. In a first approach we performed power measurements with a resistor in the VDD line of the processor core. We encountered significant disturbances within the power traces due to some additional active modules on the IHP sensor node: a Zigbee module as well as a Bluetooth module. Later it emerged that these two modules indeed caused significant fluctuations in the global power supply and hence introduced a major amount of noise in the power measurements. In order to reduce the measurement noise, we switched to EM measurements to perform localized measurements on the chip. Without any information about the actual size and position of the die within the chip’s package, we had to perform stepper EM measurements over the whole area. We chose a step size of 0.5\,mm (approximately the thickness of the EM coil used in case of the IHP node) and performed measurements at 200 positions with 20,000 traces each. At none of the positions we were able to distinctly reveal any of the key bytes. As a significant increase of traces (e.g. 1 or 2 million) multiplied with the number of positions would result in a tremendous measurement effort in terms of time and resources we decided to reduce the measurement noise and switch back to power measurements.

We disconnected the Zigbee and the Bluetooth module on the board, which considerably decreased the noise in the power traces. Furthermore, we used a more stable power supply for the processor core in order to further decrease the noise in the power line (because of external SRAM and FLASH memory modules on the board). The 32-bit processor core besides the AES module remained as last but significant noise source in our measurement setup. With this measurement setup and a 33\,Ω resistor in the VDD line of the processor we performed 6 million measurements of AES encryptions in the crypto module.

In case of AES modules, the output of the S-Box transformation is a common intermediate value used for DPA attacks. The S-Box function depends only on 8 bits of the secret key and provides a nonlinear relation between the secret key byte and the output of the S-Box operation, which significantly reduces the occurrence of ghost correlation peaks [13]. Further, we chose a very common power model, the Hamming Weight, to...
predict the instantaneous power consumption of the AES crypto module in the Leon2 processor core. It turned out that we were able to distinctly reveal 2 out of 16 round key bytes, the correlation plot of round key byte 2 is shown in Figure 17. The results show the fundamental vulnerability of the AES crypto module on the sensor node to very basic power analysis attacks, more key bytes can be revealed, for example, by simply increasing the number of measurements.

Black box power analysis attacks are quite difficult in general, because an attacker can only rely on a few basic assumptions: (1) an AES computation inevitably entails S-Box operations and (2) the instantaneous power consumption of modern CMOS circuits is related to the Hamming Weight of the processed values to some degree. A black box attacker does not exactly know when the actual cryptographic operations are performed within the device or which values run through the circuit in which order (e.g. to use a more powerful power model like Hamming Distance).

In most cases we can not assume an attacker who has absolutely zero knowledge about the attacked device. Some inconspicuous information like the size and position of the chip die could already significantly decrease the effort for performing exact EM measurements, after opening the package and identifying some basic structures on the die (e.g. memories) even very powerful localized EM measurements are feasible. An attacker could also implement more advanced attack methods like mutual information analysis.

We also performed a basic fault injection method on the IHP sensor node: the reduction of the core supply voltage in order to cause a faulty behavior of the AES crypto module. It turned out that the Leon2 processor core itself was the first module that suffered from undervoltage, which resulted in communication errors with the AES module. As we had only one IHP sensor node at our disposal, we decided to omit semi-invasive and invasive measures (e.g. de-packaging or laser-injected fault attacks) to improve our attack results.

![Figure 17: IHP sensor node, AES cryptomodule, known plaintext attack, 6 000 000 samples; round key byte 2; trace for the correct byte value is plotted in red.](image)

### 3.3.3 Attacks on AES Software Implementation

We also attacked a pure and unprotected software implementation of AES on the IHP sensor node to show the basic vulnerability of the 32-bit processor itself. The software implementation only uses native SPARC V8 instructions and the S-Box transformation has been implemented as lookup table.

We chose the S-Box operation as target of our attacks and used the Hamming Weight as power model. We performed 5 000 measurements with a sampling rate of 5 GS/s using a $33 \Omega$ resistor in the VDD line of the Leon2 processor core supply. For example, the correlation peak of key byte 11 is 0.15 which corresponds to a number of required power traces $n \approx 1 250$ (according to [47]). Figure 18 shows the correlation result of key byte 11 using 5 000 power traces, the clearly distinguishable correct key byte is plotted in red, the 255 incorrect key bytes are plotted in blue.

![Figure 18: Correlation plot for key byte 11 using 5 000 power traces.](image)
Figure 18: IHP sensor node, Leon2 processor core, known plaintext attack on software AES implementation, 5 000 samples; round key byte 11; 256 correlation traces; trace for the correct byte value is plotted in red.

Table 1 summarizes the performed attacks on the AES modules on the two sensor nodes in terms of time effort for performing the measurement, the analysis, as well as other tasks necessary to perform the attacks. The measurement and the analysis of the JN5148 wireless microcontroller could have been performed straight forward, additional time has been required to craft a suitable copper coil and for a proper positioning of the probe beneath the shield. The attack on the AES coprocessor on the IHP sensor node was quite difficult due to highly disturbed power and EM traces. A higher effort than the 110 hours has to be charged for revealing more than two round key bytes using only basic attack methods. The attack on the AES software implementation on the IHP sensor node was very easy to perform, in total only an effort of half an hour was necessary.

<table>
<thead>
<tr>
<th>Device</th>
<th>Attack</th>
<th>Meas.</th>
<th>Analysis</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>JN5148</td>
<td>EM</td>
<td>2 h</td>
<td>3 h</td>
<td>2 h</td>
<td>7 h</td>
</tr>
<tr>
<td>IHP HW-AES</td>
<td>Power</td>
<td>60 h</td>
<td>48 h</td>
<td>2 h</td>
<td>110 h</td>
</tr>
<tr>
<td>IHP SW-AES</td>
<td>Power</td>
<td>0.2 h</td>
<td>0.3 h</td>
<td>-</td>
<td>0.5 h</td>
</tr>
</tbody>
</table>

Table 1: Summary of the attacks on the AES modules in the sensor nodes.

3.3.4 Attacks on IHP Sensor Node - Hardware ECC

Elliptic Curve Operations
Elliptic Curve Cryptography (ECC) is applied in a variety of protocols which take use of the one-way property of the elliptic curve point multiplication (ECPM). The ECPM is a multiplication of a point \( P = (P_x, P_y) \) on the two-dimensional elliptic curve with a factor \( k \). This operation, sometimes referred as kP-multiplication, can be computed straightforward, while to inverse operation is computationally infeasible. The IHP ECC accelerator executes the ECPM.

In practice ECC is often applied as basis operation for the key exchange and agreement (Elliptic Curve Diffie-Hellman (ECDH)) and signature algorithms (Elliptic Curve Digital Signatures Algorithm (ECDSA) [39]). For ECDSA \( P \) is the secret, while in ECDH the secret relies on \( k \).

Therefore both \( k \) as well as the coordinates of \( P \) should be protected. In the following a set of attacks is presented which allow to extract the secret information from the circuit.

Reading the registers
Certainly the most straightforward method to extract the secrets is to read the keys as they are stored in the
Figure 19: Detail of the power trace of the ECPM operation executed on the ECC accelerator on the IHP node.

registers of the ECC block. This can be executed from any master device on the AMBA bus, including the microcontroller and the debug interfaces.

**Timing Attacks**

In Section 3.1.1 timing attacks were introduced. In the following we apply these attacks to extract information from the circuit. This attack is performed with the knowledge that the implemented algorithm to compute the ECPM processes the factor $k$ while the time for computing a ‘1’ is slightly different to the time needed for a ‘0’ (56 to 54 clock cycles). Figure [19] shows the power trace of the ECC component. To improve the visibility the traces were obtained in simulations with the Primepower tool [67].

We perform two practical experiments: one monitoring the time for the entire operation, and a second that measures the time for processing the individual bits of $k$. Monitoring the entire operation does not need additional equipment, since all information (time and start and end of the operation) are visible on the microcontroller. Depending on the protocol implementation they are even visible on a remote side.

The goal of the attack is to extract the number of ‘1’s in the factor. This number is referred as $x$ is the following equation that states how the measured time is composed.

$$x \times t_1 + (232 - x) \times t_0 + t_{fix} = t_{measured}$$

$t_1$ and $t_0$ are the number of clock cycles needed to compute one bit for a ‘1’ and a ‘0’, respectively. $t_{fix}$ represents the constant amount of time for at the beginning and end of the point multiplication. This part does not vary with data and key. $t_{measured}$ is the time (in number of clock cycles) measured for the complete ECPM. This equation can be resolved to $x$ to obtain the number of ‘1’s in the factor:

$$x = \frac{t_{measured} - t_{fix} - 232t_0}{t_1 - t_0}$$

In the tested implementation $t_{fix}$ is 433, $t_0 = 54$, and $t_1 = 56$. Considered we measure $t_{measured} = 13051$, the resulting number of ‘1’s in $k$ is $x = 45$.

This information does not break the key, but reduces the search space significantly.

In the second experiment we zoomed into the power traces of the ECPM operation to identify the duration of the computation for each bit. As it can be seen in Figure [19] the rhythm of the bit processing can be easily recognized. It allows to determine the duration for each clock cycle precisely. The result of this attack is the extraction of the factor $k$ directly without additional computation. However, contrary to the timing attack that monitors the entire operation, this attack needs physical access and additional equipment.

**Power Analysis**

The trace in Figure [19] already indicates that timing attacks are not required if the power traces can be obtained.
The differences between the computation of a ‘1’ and a ‘0’ can be recognized with the naked eye. In fact we first analyzed the power traces to identify the cycles for each bit. By this also a power analysis reveals $k$ directly. While not tested, it can be assumed that the same information can be extracted with electromagnetic measures.

**Discussion of the timing vulnerability**
From the attacks described above it is observable that the current IHP ECC design is somewhat vulnerable against timing and power attacks. However, the attacks only allow to derive the integer $k$ of the $kP$ ECPM. For most ECC-based protocols (such as ECDSA and ECDH) this $k$ is random and is supposed to be used only one time. The knowledge of $k$ would compromise the secrecy of the operation, but usually does not break the system.

The actual secret of many algorithms is stored as a point $P$ on the elliptic curve. The coordinates of this point do not influence the run time of the computation and could not been observed as influencing the power traces in a way that allows to extract that information.
4 Attacks on System Level

4.1 Background: Software Attacks on General Purpose Computers

4.1.1 Code Injection Attacks

Code injection attacks are common on general purpose computers and count among the most dangerous attacks on a system. If an attacker is able to inject arbitrary code in a system, he is able to perform any possible actions at the current privilege level. Those attacks rely for example on:

- using social engineering to trick the user into executing a malicious program,
- opening a document that embed malicious scripts,
- abusing a update mechanism [16],
- improper checks on user supplied data,
- abusing of software vulnerabilities.

Here we focus on the abuse of software vulnerabilities. One of the first widespread use of such attacks is the Morris worm (also known as the Internet worm) [66, 56]. The Morris worm spread on the Internet during winter 1988. The Internet was composed of only a few thousands nodes at that time but the spread of the worm was very fast and it disrupted an important part of the network. The worm was active for a few days before being stopped [65]. The analysis of the worm showed that, among several infection techniques used, it performed a stack-based buffer overflow that exploited a vulnerability in the finger daemon in order to inject code on the stack. This injected code was then executed from the stack and launched a shell, which gave full control of the computer to the worm.

In this section we describe common techniques used for code injection attacks that abuse software vulnerabilities. In Section 4.2 we describe common counter-measures as well as techniques used for detection of such attacks.

Buffer Overflow A buffer overflow condition (also known as buffer overrun) occurs when data is written to a memory allocated region which is not large enough to contain the data. If proper boundaries check are not in place to prevent the overflow, memory regions contiguous to the overflowed buffer will be corrupted. The possibility and consequences of exploiting the overflow depends on the location of the overflowed buffer.

```c
// ...
char src[]="ABCDEFGH"
char tmp_buff[5];
int password_checked;
// ...
strcpy(tmp_buff, src);
// perform some action on backup string tmp_buff
// ...
```

Figure 20: Simple string based buffer overflow vulnerability.

![Memory layout after the buffer overflow](image)

Figure 21: Memory layout after the buffer overflow presented in Figure 20
There exists a set of very well known functions or coding techniques \cite{55} that are unsafe and often leads to buffer overflows. For example, string manipulations that rely on the presence of a NULL byte at the end of the string are subject to buffer overflow. Such standard functions do not check the length of the string but instead rely on the NULL byte to detect the end of the character chain. Figure \ref{fig:strcpy} shows a code that performs a string copy using the unsafe `strcpy` function. The data copy ends only when a NULL byte is found in the source string however the source string is longer than the allocated destination variable. Figure \ref{fig:mem_layout} shows the resulting memory layout with the characters `FGHI` written after the end of the dedicated memory region for the `tmp_buff` variable. On a general purpose computer if this memory region is not in a mapped memory page this will result in a segmentation fault error. However, if the overflow remains in a valid memory page it will likely overwrite another variable.

The position in memory of the overflowed buffer is crucial to the ability of an attacker to exploit it for malicious purposes. In the following sections we show how this can be used to perform malicious actions.

**Stack-based Buffer Overflow: Control-Flow Manipulation Using a Buffer Overflow**

Functions and procedures are basic building blocks of programming languages, they embed code that implement an action in an independent block. Functions are called with a call instruction that diverts the control flow to the top of the function code. Upon completion the execution is returned to the caller with a return or ret instruction (Figure \ref{fig:call_ret}). During the call instruction the address to return to (i.e. the address of the instruction following the call instruction) is saved on the stack, this same address is retrieved from the stack by the return instruction.

On most microprocessors a unique stack is used to store control flow information as well as other data. Each frame of the stack usually contains the following data:

- saved return address of the caller;
- function variables and parameters;
- saved register values, according to the specific Application Binary Interface (ABI).

Implementation details vary across different architectures, Figure \ref{fig:stack_layout} depicts a simple layout example for a portion of the stack. Control flow information, such as return addresses, are stored alongside other function data.

When a buffer overflow occurs on a buffer allocated on stack the attacker is able to overwrite part of the stack. One of the most interesting part of the stack for an attacker is the return address saved during a function call. This return address is used when the function executing ends (i.e. a return instruction will be executed) to move the program counter to the code where the function was called. However, if this return address was maliciously modified with a buffer overflow the attacker has gained full control over the program counter.

Buffer overflows that occurs on a variable not allocated on stack can also lead to control flow manipulation. A common example of such an attack is when a buffer allocated close to a function pointer is overflowed. With such an overflow the attacker can modify the value of the function pointer. Latter, when the function is called from this pointer the control flow will be redirected to the code of the choice of the attacker.

**Redirecting Execution on Stack**

In its most basic form a stack-based buffer overflow is used to inject instructions (i.e. the payload or shellcode) on the stack and redirect the control flow to those instructions by modifying the return address. This attack becomes more difficult when the attacker is unaware of the current stack pointer or address where the instructions were written. When this address is not accurately known the
attacker either needs to guess the address, which can be very slow, or needs to use other techniques such as using a **NOP sledge** or finding **trampolines** [64].

A NOP sledge is a long sequence of instructions, that performs no operations, which is inserted before the actual injected instructions. When the attacker has an approximate guess of the position of his injected code, he is able to redirect execution to an address in the NOP sledge. The processor will then execute the NOP instructions until the actual payload is reached. Therefore, the attacker does need to know exactly the address where the payload has been written, only an approximate knowledge is enough to redirect execution in the **NOP sledge**.

Another common technique used to execute code on the stack is the use of **trampolines**. An attacker will locate an instruction such as `jmp esp` or `call esp` (Intel x86 assembly) that directly redirects execution to the stack. If such an instruction is found at a fixed address he will use this address to overwrite the return address on stack. This will lead to execute the trampoline which will in turn redirect execution on the payload stored in the stack.

### 4.1.2 Malicious Code Execution Without Code Injection

**Return to libc : Redirection to Existing Functions** The previously described technique assumes that the stack (or other memory region writable by an attacker) is executable. However, this is not the case with modern operating systems that provides defenses against execution of code on any writable section (described in Section 4.2.2). It is also impossible to execute instructions from the stack on Harvard architecture processors as we will describe later.

Several techniques have been therefore developed to bypass these protection mechanisms. One of the first public technique was the return to libc (Also known as return-into-libc) attack [62] where the attacker does not inject code to the stack anymore but instead executes a function present in the address space. As on UNIX systems the C library (libc) is loaded for most programs in order to use basic functions of the C library, it is convenient to use the libc as a target of the return to libc attack. Moreover, the C library contains interesting functions for an attacker, the most common function called in a return to libc are the `system` or the `exec` function.

The return to libc attack, when it uses a stack-based buffer overflow, usually consists in writing data to the stack and overwriting a return address on stack. This address is modified to point to a function, at a known location, which will be called when the exploited function returns. When this function is called it will look for parameters on the stack, and use the data that was previously written by the attacker during the buffer overflow. Therefore, the attacker is able to execute any function and pass parameters to it. The most commonly used
functions are the exec or system functions, to which is passed an argument that spawns a shell or open a server socket on the system under attack. Using those functions, and being able to pass arbitrary parameters to it, it is easy to launch a shell or open a socket for latter connection. Subsequently the attacker can connect to this socket and obtain a prompt, he has full control over the system.

**Borrowed Code Chunks**  As seen above, the return to libc technique works well when the functions called by the attacker do not need parameters or when the Application Binary Interface (ABI) requires parameters to be passed on the stack. However, if parameters needs to be passed in registers this attack can’t work directly as the attacker’s data is only present in the stack. This is the case in the 64-bit Intel architecture. The borrowed code chunks technique was developed as an enhancement to the return to libc attack to load parameters to registers. The main idea is to craft a payload that will chain code present in the application address space (e.g. application code or libraries) to load proper values from the stack to registers. Once those values have been moved to registers the function can be executed (e.g. “returned to”) with the parameters loaded in registers.

As the code chunks are carefully selected to contains a few instructions and terminate with a return instruction, it is possible to chain them. To chain those code chunks, the attacker needs to build a stack layout that contains the data that will be used by the code chunk (e.g. when a pop instruction is encountered) as well as the return addresses that points to the next code chunk. Therefore, by chaining the code chunks together it is possible to write an attack payload that perform more complex attacks.

**Return-Oriented Programming**  The “return-into-libc” and code chunks borrowing attacks have been extended into a more generic attack. Return-Oriented Programming generalizes this technique and defeats systems that prevents execution of code in writable memory regions by executing preexisting sequences of instructions to perform arbitrary computations. Group of instructions terminated by a return instruction, called Gadgets, are first located in the process address space. Gadgets are performing actions useful to the attacker (i.e., pop a value in stack to a register) and returns to another gadget. The objective of the attacker is to find a Turing complete gadget set. A Turing complete gadget set can be used to build a Turing machine and therefore the attacker can chain those gadgets by controlling the stack to perform arbitrary computations. While this was first demonstrated on the Intel x86 architecture, it was further demonstrated to be possible on the SPARC architecture. Once a Turing complete gadget set is available it is possible to build a compiler to automatically generate return-oriented programs.

**4.1.3 Non Buffer Overflow-Based Software Attacks**

Many different techniques are used to launch software attacks. We previously detailed the techniques used during starting with a buffer overflow. This section describe other sources of control flow manipulations.

**Stack Overflow**  A stack overflow is an event that occurs when the stack usage grows until it reaches and overlaps with another section of memory. This is the definition we will use throughout this section and it must not be confused with stack-based buffer overflow. As seen in Section 4.1.1 the latter is the consequence of a vulnerable or malfunctioning program (e.g. improper boundary check) and the former is the consequence of an out-of-memory condition.

Stack overflow is an out of memory condition common in embedded systems with highly constrained memory availability. This, for example, means that the stack overflow can occur with a correct program or a program written in a type or memory safe language. When this happens on a general purpose computer the situation is detected thanks to guard pages that limits the stack growth. However, this is a limited defense as, in some cases, the guard page can be “jumped” over.

**Other Sources of Control-Flow Manipulation**  Any software vulnerability that can allow an attacker to write memory at arbitrary position can lead to a control flow attacks. With an arbitrary memory write an attacker can modify a return address or a function pointer to manipulate the control flow. Improper string format usage

---

1. AMD64 or Intel x32_64
2. such as the $W \oplus X$ technique, this is introduced in more details in Section 4.2.2
in functions such as a `printf` that let the attacker manipulate the format string as well as heap data structures corruption could allow such arbitrary memory writes.

### 4.2 Mitigation Techniques on General Purpose Computers

As new attacks techniques became public defenses have been developed in order to make exploitation difficult or to prevent the attacks. Any solutions that could prevent or complicate any of these operations could be useful to mitigate attacks. However, when a new attack is made public it is often immediately used, on the opposite new defensive techniques that are proposed takes often years to be integrated in real systems, if ever. This observation leads to two different approaches for defensive techniques. The ideal case is when a defensive technique not only prevents one kind of attack but a larger set of attacks, it would be more likely to prevent abuse of future attack methods. This is often an idealistic view but it is working in some cases. An example of such a defense that prevents (or make more difficult) the exploitation of control flow attacks is the address space randomization-based defenses. For example, it prevents straightforward exploitation of return to libc but also the return-oriented programming that was introduced after ASLR-like techniques became present in most operating systems.

#### 4.2.1 Preventive Measures

**Memory Safety**  Most of the malicious code execution attacks presented in previous section have as primary source the lack of strong variables boundary checks or type enforcement. This lack of enforcement is present in many low level or weakly typed programming languages. While languages that automatically prevent such attacks are widespread, such as Java, unsafe languages are still of a widespread use. Moreover, even with strongly typed languages related attacks are possible. For example, Java virtual machines are themselves implemented in C language and rely on many libraries not implemented in Java that may have flaws. Flaws in the Java virtual machine have already been shown to be exploitable [26]. Moreover, implementation errors of the Java specification in the virtual machine can also lead to bypassing the memory protections or type checking enforcement [44, 48].

An alternative solution is to provide extensions to unsafe languages in order to add extra checks on memory accesses and manipulations. In Deputy [20], authors propose to annotate the source code of a C program with extra information on constraints that must be enforced during run-time on variables (e.g. the array A is not bigger than the value contained in variable X). Those annotations allow the compiler to add additional checks on the validity of the variables before use. Furthermore, in Safe TinyOS [21], Cooprider et al. did extend such scheme to the NesC language which is used in by applications for the TinyOS operating system, it is now part of the main TinyOS branch. The drawbacks of such an approach is that the annotations must be correctly written and if they are omitted a memory safety violation might not be caught by the system.

**Control-Flow Integrity**  There is a wealth of different proposals on how to solve control flow vulnerabilities. In Control Flow Integrity [4], Abadi et al. propose to embed additional code and labels in the code, such that at each function call or return additional instructions additional code is able to check whether it is following a legitimate path in a precomputed control flow graph. If the corruption of a return address occurs that would make the program follow a non legitimate path, then the execution is aborted as malicious action or malfunction is probably ongoing. The main drawback of the approach is the need for instrumentation of the code, although this could be automated by the compiler tool-chain, it has both a memory and computational overhead. Similar approach as been proposed for wireless sensor networks devices based on the AVR processor [27].

#### 4.2.2 Protecting the Stack

**Protecting the Return Addresses on Stack With Canaries**  Stack protections, such as random stack canaries, are widely used to secure operating systems [22][11]. The random stack canary is usually implemented in the compiler with operating system support. When compiling a function, the compiler generates additional code in the prologue and the epilogue of each function. The prologue places a value, called a canary, between the return pointer (and the frame pointer if present) and the local function variables (Figure 24). The canary is checked for validity in the epilogue of the function before returning execution to the caller function. If the
canary value has changed, this is an indication that an abnormal operation occurred. This usually indicate that a memory corruption occurred, such as a stack-based buffer overflow. If the canary value has been detected to be modified the epilogue code of the function does not return to the caller (i.e. with the value stored on the stack) as this value is likely to have been corrupted as well. When such memory corruption occurs the control is passed to a specific code that will take appropriate measures. Usually this is a function that will abort execution and log an alert message.

This technique prevents straightforward return address overwriting, such as stack-based buffer overflows and stack overflows. However this technique has some drawbacks. First, canaries add extra instructions to be executed at each function call thus introducing non negligible overheads. Second, canaries have been shown to have a number of vulnerabilities [6], for example if the attacker is able to use a double memory corruption, that corrupts a pointer and later writes a value to the address it points to. In such case the attacker is able to start writing after the canary value, and therefore corrupt the return address while avoiding detection.

Preventing execution on stack

On general purpose computers, in order to prevent buffer overflow attack that execute code injected on stack, memory protection mechanisms, known as the no-execute bit (NX-Bit) or Write-Xor-eXecute ($W \oplus X$) [7, 24, 70, 52] techniques have been proposed. These techniques enforce memory to be either writable or executable, but never both. Therefore this prevents code to be executed from the stack or other writable memory areas. For example the section of memory that holds the code of the application and where the shared libraries code is mapped will be marked as executable, but will not be modifiable. While the stack, heap and data sections (BSS or DATA) will be marked as modifiable but not executable. Therefore, if the $W \oplus X$ technique is enabled an attacker would still be able to inject code in the stack (or other sections that are modifiable) but will not be able to execute his code. Usually trying to execute instructions in a page marked as non executable will generate an exception from the memory management manager of the operating system. While those techniques first appeared as non official patches for operating systems [70], they are now part of most operating systems and hardware support has been introduced.

4.2.3 Making Exploitation of Control-Flow Attacks Difficult

Address Space Layout Randomization

Address space layout randomization [71] can hinder control flow attacks. It is a technique where the base addresses of various sections of a program memory are randomized before each program execution. ASLR (Address Space Layout Randomization) [71] randomizes the address of the loaded binary code as well as the memory used for data sections (such as stack, heap, data and BSS sections). This randomization do not prevent buffer overflows or return address corruption, it makes its exploitation more difficult. It helps to protect against control flow attacks as an attacker do not know in advance the address where code or functions are located.

However, in [60] Shacham shows that the effectiveness of address-space randomization is limited on 32-bit architectures by the number of bits available for address randomization, which may not stop an attacker with
the possibility to perform multiple attempts. Additionally, the fork system call, that spawns a new process, is commonly used by network server software, during a fork the child process isn’t randomized again. Therefore, an attacker can use the knowledge of the randomization of one process to attack child processes.

This limited randomness problem would be even more severe on embedded systems that typically have a 8-bit or 16-bit address space.

In an extension of ASLR, ASLP \[41\] (Address Space Layout Permutation) proposes to improve ASLR by randomizing the binary code itself. By modifying the layout of the binary itself, it is possible to improve the number of bits of randomness in the address of the portions of code an attacker would use. This adds an extra layer of difficulty to guess the addresses of interesting functions or code chunks.

**Eliminating the Call Stack** In \[74\] Yang et al. introduce a source to source transformation that translates traditional functions calls into a flat program without function calls. The transformation is similar to function in-lining without the usual code size overhead. The overhead is avoided as functions are lined once and called not as usual functions but with a jump to a label.

Additionally, the variables that were allocated on stack are now statically allocated on the BSS section. A straightforward implementation would be very memory consuming. However, it has been shown that as the variables that use to be allocated on stack are not used simultaneously. Therefore, optimization can be performed that limits the memory overhead.

The advantage of this technique is that as functions are in-lined an attacker that overflows a buffer can’t overwrite a return address as such return address is not present anymore. Moreover, if control flow corruption occurs, the likelihood for an attacker to find sequences of instructions terminated by a return instruction is very small, as almost no functions remain after program flattening.

The main limitation of this technique is that the transformation needs to be performed at source level and therefore requires a complete recompilation of the program. Flattening cannot be applied to binary libraries or existing programs. Moreover, Interrupt handlers cannot be flattened as their call site and return address cannot be known in advance. Such interrupt handlers could be maliciously used as, just like functions, they have to end with a return instruction. This technique first appeared for wireless sensor nodes, its feasibility for large software stack present in commodity software has yet to be demonstrated. For example it is unlikely that shared libraries could still be used with such a technique. To avoid shared libraries, that contain functions called from programs, it would be required to completely in-line all the used functions from source, this would have serious performance impact on general purpose computers as the advantage of shared memory pages and libraries would be lost, programs would be much larger. However it is well suitable for embedded systems.

**4.2.4 Protection by Modification of The Stack Model**

**Return Stack** In \[72\] the authors present StackShield that uses a compiler supported return stack. The compiler inserts a header and a trailer to each function in order to copy to/from a separate stack the return address from/to the normal stack.

In \[75\] Younan et al. propose to split the stack into multiple stacks depending on the kind of data that has to be stored on the stack. For example return addresses and function pointers are stored on a dedicated stack, arrays of characters will be stored on another stack and arrays of pointers will be yet in another stack. The approach proposed leads to five separate stacks each of them being allocated in sequence but separated from each other by a guard page. A guard page is a page of memory that is intentionally left non allocated, any attempt to write to this page, for example during a buffer overflow, will lead to a page fault exception which will be handled by the kernel. The kernel will therefore detect buffer overflows. The drawbacks of this approach is that it requires a memory management unit, which is unavailable on constrained embedded devices.

Those both approaches are implemented at the compiler level and therefore no backward compatibility of preexisting software is possible without access to the source code. The programs need to be re-compiled with this modified compiler. Moreover, as additional instructions are introduced there is non negligible a computation and memory overhead. In \[29\] the stack is used as usual but return addresses are saved on a separate stack which can only be accessed by call and ret instructions. With additional addresses checking this prevents both overwriting of return addresses and stack overflows (but not buffer overflows, which is a separate problem).
**Hardware-based Approaches for Return Stacks** In [73] the authors propose a return stack mechanism where dedicated `call` and `ret` instructions store and read control flow information from a dedicated stack. However, the only guarantee for this return stack integrity is that it is located far away from the normal stack. This does not prevent modification of the return stack, it just makes it more difficult. Double corruption attacks [6] would allow an attacker to corrupt a data pointer first and then modify an arbitrary memory location on the return stack.

### 4.2.5 Malicious Code Detection

**Hardware-based Detection** If an attack can not have been prevented or detected it is important to be able to detect the presence of malicious code. Various approaches have been taken. The most widespread standard on general purpose computers is the Trusted Platform Module (TPM). A TPM is a small independent device usually attached to the main board of a computer. This chip is dedicated to performing attestation of software. When the computer starts the TPM attest each layer of the operating system, starting from trusted code in a read-only part of the BIOS. Each subsequent piece of software is checksummed, this checksum is verified against a trusted version of the checksum present in the TPM. If the checksum is valid, the next piece of software can be executed.

A software and hardware architecture has been proposed in [37] that shows the feasibility of attestation using a TPM device on wireless sensor networks devices. However, the solutions based on a TPM attests only the software during the boot of a device. If the device is compromised after the boot, for example with a code injection attack, the TPM can’t help to detect this attack before the next reboot.

**Software-based Detection** Software-based attestation on general purpose operating systems [40, 58] has been previously proposed. The idea is to provide an environment and specific self-checking code that prevents an attacker from modifying the running software. Therefore if an attacker modifies the self checking code or another part of the code the checksum result will either be wrong or delayed. In both cases the attack is expected to be detected. However [40] has been showed to have serious weaknesses [61]. In next section we will detail several schemes dedicated to embedded systems, and more specifically for Wireless Sensor Network devices.

### 4.3 Software attacks on WSN nodes

#### 4.3.1 Generic Attacks on Microcontrollers

Traditional buffer overflow attacks usually rely on the fact that the attacker is able to inject a piece of code into the stack and execute it. This exploit can, for example, result from a program vulnerability such as a stack-based buffer overflow as described in Section 4.1.1.

In the Von Neumann architecture, a program can access both code (TEXT) and data sections (data, BSS or stack) without distinction. Furthermore, instructions injected into data memory (such as stack) can be executed. As a result, an attacker can exploit buffer overflow to execute malicious code injected by a specially-crafted packet.

In Mica-family sensors, code and data memories are physically separated in two distinct address spaces. The program counter cannot point to an address in the data memory. The previously presented injection attacks are therefore impossible to perform on this type of sensor [32, 31]. This results in a natural defense which is similar to that of systems with $W \oplus X$.

Furthermore, sensors have other characteristics that limit the capabilities of an attacker. For example, packets processed by a sensor are usually very small. For example TinyOS limits the size of packet’s payload to 28 bytes. It is therefore difficult to inject a useful piece of code with a single packet. Finally, a sensor has very limited memory. The application code is therefore often size-optimized and has limited functionalities. Functions are very often inlined. This makes “return-into-libc” attacks [62] very difficult to perform.

Because of all these characteristics, remote exploitation of sensors is very challenging, the few next paragraphs describes some of the existing work in this domain.

**Stack Execution on Von Neumann Architecture Sensors** In [30, 31], Goodspeed, describes how to abuse string format vulnerabilities or buffer overflows on the MSP430-based Telosb motes in order to execute mali-
cious code uploaded into data memory. He demonstrates that it is possible to inject malicious code byte-by-byte in order to load arbitrary long bytecode to overcome the packet size limitation. As Telosb motes are based on the MSP430 micro-controller (a Von Neumann architecture), it is possible to execute malicious data injected into memory. However, as discussed in Section 4.3.1 this attack is impossible on Harvard architecture motes, such as the Micaz. Countermeasures proposed in [31] include hardware modifications to the MSP430 micro-controller and using Harvard architecture micro-controllers. The hardware modification would provide the ability to configure memory regions as non executable. In our work, we show by a practical example that, although this solution complicates the attack, it does not make it impossible.

**Level of difficulty:** Easy  
**Chances of success:** High  
**Usual countermeasure:** Medium, $W \oplus X$

**Mal-Packets** In [35], Gu and Noorani shows how to modify the execution flow of a TinyOS application running on a Mica2 sensor to perform a transient attack. This attack exploits a buffer overflow in order to execute gadgets, i.e., instructions that are present on the sensor. These instructions perform some actions (such as modifying some of the sensor data) and then propagate the injected packet to the node’s neighbors. While this attack is interesting, it has several limitations. First, it is limited to one packet. Since packets are very small, the possible set of actions is very limited. Second, actions are limited to sequences of instructions present in the sensor memory. Third, the attack is transient. Once the packet is processed, the attack terminates. Furthermore, the action of the attack disappears if the node is reset.

**Level of Difficulty:** Difficult  
**Chances of Success:** High  
**Usual counter measure:** Difficult, Address Space Randomization  
**Secondary counter measure:** Medium, Return address protection

**Return-Oriented Programming** [28] presents a code injecting attack on a Harvard architecture device (Atmel AVR). Harvard-based devices have an inherent protection against code injection, because data and program memories are physically separated and belongs to a separate address space. In [28] after exploiting a memory corruption bug a sequence of instructions terminated by a return instruction are chained from the stack. This sequence of instruction controlled from the stack is known as return oriented programming [54] (or Code borrowing attack [43] without Turing completeness). In [18] return oriented programming was used to change the behavior of a voting machine without performing a code injection attack.

**Level of Difficulty:** Difficult  
**Chances of Success:** High  
**Usual counter measure:** Difficult, Address Space Randomization  
**Secondary counter measure:** Medium, Return address protection

**Stack Overflows on Microcontrollers** Stack overflows are common on simple micro-controllers, due to their limited memory size. This condition can occur, for example, when too much data is allocated on the stack or when the depth of the stack grows too large. In both cases, the stack exhausts its available memory and overlaps with other memory sections like the BSS section.

This is both a reliability problem and a security problem. It is a reliability problem as the stack overflows in other memory regions, it can corrupt the data stored there. This usually leads to bugs that are difficult to track. Because, for example, the corrupted variable will depend on the layout of variables in the BSS section. This therefore depends on how the compiler will order variables in memory. A slight change in the program might lead to a different layout of variables and move the corruption to another variable. This could give a false belief that the problem is solved. Another difficulty with stack overflows is that the corruption can occur on very rare events (e.g. an interrupt occurs at the exact point when the stack usage is maximal), and therefore leads to problems that are difficult to track and reproduce.
Figure 25: Memory layout before a stack overflow, the stack and the BSS sections are not overlapping.

Figure 26: Memory layout during a stack overflow, the stack is overwriting the BSS section. The variables in BSS section are corrupted. If a write is performed to the BSS section during the overflow a return address can be modified, an attacker could take advantage of this.
It can be a security problem as an attacker might take advantage of a stack overflow to overwrite a return address without any specific program vulnerability. When this function will return the control flow will be directed to the address chosen by the attacker.\footnote{While, we are not aware of any practical example of such an attack on embedded systems, this has already been performed abusing the \texttt{alloca()} function \cite{39} on general purpose computers \cite{45}.}

Stack overflow conditions are easily detected in general purpose operating systems where a page fault occurs when memory is accessed beyond the currently allocated stack space. However the lack of MMU make this impossible to implement on constrained embedded systems.

In embedded systems the stack consumption can be analyzed before execution performing static analysis on the program \cite{51}. Static analysis will reveal whether the device will have enough memory to execute the application. However in some cases it can be difficult to know exactly the maximum stack consumption, for example:

- when indirect calls are present the tool has to perform data flow analysis, which is not always feasible,
- when re-entrant interrupts are used the call depth could be unbounded,
- if recursive function calls are performed, data flow analysis would have to be performed, if possible.
- some compilers implement a way to allocate dynamic memory on the stack as non standard extensions, for example \texttt{gcc} provides the \texttt{alloca()}\cite{69} built-in function for this purpose. This is again a difficult case for static analysis tools.

When such features are used in a program it is impossible to perform abstract interpretation (unless a full control and data flow graph can be generated). In such cases specific run-time mechanism should be used.

### 4.3.2 Hardware/software Attacks Specific to Microcontrollers and WSN Nodes

In this section we describe a few attacks that are mostly specific to embedded systems and involve both hardware and software interactions with the device, they can therefore be considered as hybrid attacks. Those attacks can therefore usually not be performed remotely. However, while they require physical interaction with the device in addition to a software attack they can be performed without damaging the device, i.e. we do not consider invasive attacks, that would require de-packaging and tampering with the device (e.g. using a FIB).

#### Attacks via JTAG

The easiest way to attack or gain control over a node would be to use JTAG. The IEEE 1149.1 JTAG standard is designed to assist electronic engineers in testing their equipment during the development phase. It is also used in current equipment for on-chip debugging, including single-stepping through code, and for reading and writing memory \cite{3}. Majority of WSN motes offer JTAG Test Access Port (TAP) including Atmel and Texas Instruments microcontrollers. Just as the developer, an attacker can easily gain control of the node with JTAG if the access is open. An attacker could read and control:

- Memory Address Bus
- Memory Data Bus
- CPU
- Non-Flash Memory
- Flash Memory

And more, to launch such an attack the attacker does not require any special skills or equipment except for an appropriate adapter cable, portable computer and MSP430 Programming via the JTAG Interface User Guide where in-depth detail about programming via the JTAG can be found. By simply following the instructions that are given, an attacker can easily extract the cryptographic keys from the external flash memory to gain further access to a network or malfunction a node, of course if the aim of the attacker is to just malfunction a node, the easiest way would be is to write unlisted values to the JTAG instruction register of the node, a list of defined
Bus Tapping  Most sensor nodes that are available in the market, use peripheral devices such as radios or EEPROM or external flash, interfacing with microcontrollers through SPI, I2C or other standard bus interfaces.

This is a major vulnerability in sensor node, as any one with a logic analyzer or an oscilloscope can easily values for JTAG instruction register could be found in the user guide. Even if there is no JTAG connector provided on the circuit board, an attacker can still get access to the JTAG ports by directly connecting to the right pins on the microcontroller which can be looked up in the datasheet. The best counter-measure to this problem would be to disable the access to the JTAG. For example, one can disable the JTAG in Texas Instruments’ MSP430 by programming (or blowing) the security fuse. Once the security fuse is blown, future access to the MSP430 via the JTAG interface is impossible as the JTAG interface is permanently disabled by this procedure.

Level of difficulty: Low
Chances of success: Medium
Usual countermeasure: Easy, deactivate JTAG in production devices

Figure 27: Typical WSN device, PCB with many discrete components, an attacker can easily eavesdrop communications between chips.
Figure 28: Typical PCB bus tapping to recover firmware or key material. Here dumping the serial EEPROM of a TI EZ430U Rev 2.0 board (Picture © T. Goodspeed).

eavesdrop. All an attacker would have to do is connect two electrical probes constructed from medical syringes or the ones that come along with the logic analyzer on to the serial interface between the microcontroller and the EEPROM/external flash/radio as show in Figure 28.

By capturing the data transmitted over this interface, an attacker is able to observe all the communications between the two peripherals, capturing radio configuration information, cryptographic keys, network authentication credentials and other sensitive data [3]. Alternatively, an attacker could manipulate the target network by injecting new packets onto the bus.

The attacker needs to make a small investment to carry out such an attack, a low end logic analyzer from Saleae costs as low as US $150.

In a more sophisticated attack described in [3] the attacker connects a second microcontroller to the I/O pins of the flash chip. The attacker would wait that the original microcontroller do not access the data bus to perform the attack, if this is the case the attack will be completely unnoticed.

In some conditions, the attacker can sever the direct connection between the mote’s microcontroller and the flash chip, and then connect the two to her own microcontroller. The attacker device would then simulate the external memory to the mote, making everything appear unsuspicious.

The reason this type of attack should be considered a major threat to the network is that it does not disturb the normal operation of the sensor node, hence we would never notice such an attack and it can go on for a long time.

Another variation of this attack described by the authors of [1] is once the attacker successfully connects to the mote with his/her own mote the attacker could to a ‘mass erase’ of the mote microcontroller and then put his/her own program to read the external memory contents, as we will see a little later to do a mass erase there is no need to know the BSL password. But this attack is not a passive attack and hence it could be noticed as it stops the normal operation of the mote.

Countermeasures against such attacks are:

- using encryption between the devices, i.e. to prevent any exploitable information to be exchanged on
PCB wires.

- perform sensitive operations on the main chip, for example while it is common to have the AES engine on the radio chip (e.g. CC2420) this leads to the need to transmit the keys and non encrypted data over the bus on PCB. An alternative at design time is to embed the AES engine on the main chip.

- at design time embedding all necessary devices on the same chip.

**Level of difficulty**: Low  
**Chances of success**: Medium  
**Usual countermeasures**: Medium, at chip design time embed devices in one package  
**Secondary countermeasures**: Medium/difficult, encrypt data on bus

**Level of Difficulty for [3]**: Medium  
**Chances of success**: Medium  
**Usual countermeasures**: Medium/difficult, at chip design time embed devices in one package  
**Secondary countermeasures**: Medium/difficult, encrypt data on bus

The **MSP430 Bootloader** If the JTAG interface is blown, the only other way to program or read a MSP430 is through Bootstrap loader (BSL). In BSL, there are two types of operations, password protected operations and unprotected operations. Most of the important commands such as read and write come under password protected operations. There are 4 methods that an attacker can attack a mote with BSL have been mentioned here. Here are the commands are protected or unprotected (some of them are present in specific versions only):

- **Unprotected Commands**:
  - Receive password
  - Mass erase (erase entire flash memory, main as well as information memory)
  - Transmit BSL version
  - Change baud rate

- **Password Protected Commands**
  - Receive data block to program flash memory, RAM, or peripherals
  - Transmit data block
  - Erase segment
  - Erase check
  - Set Memory Offset
  - Load program counter and start user program
  - Change baud rate

**MSP430 Bootloader: Method 1** The BSL password is not defined by user, it is rather defined by the compiler which places it in the interrupt vector table (IVT); therefore the compiler creates a unique password for each different source code. The password is 16*16 bit value and consist of the flash memory content at addresses 0xFFE0 to 0xFFFF. Immediately after a mass erase operation, the password is reset (32 bytes with the value 0xFF). One choice for an attacker is to try guessing the password. An obvious method to guess the password is brute force. There are two types brute force methods that have been proposed, but none of them are feasible as the time require is too long [32] (est. 32 years[32] or 128 years [3]).
Level of difficulty: Low  
Chances of success: Low  
Usual countermeasure: Easy, randomize Interrupt vector Table [12]

MSP430 Bootloader: Method 2  This method was developed by Travis Goodspeed, it is based on side-channel attack [33]. This method is only valid for specific versions of the BSL. Those versions have a password comparison routine which suffers from unbalanced timing that is processing an incorrect password takes two clock cycles longer than a correct byte. By observing the external timing, it is possible to determine the correctness of individual bytes, drastically reducing the amount of time required to guess a password. To make it even easier to launch these attacks, Goodspeed designed a device that can measure these timings, the BSLCrack [33]. An attacker using this method to recover BSL password, after checking the BSL version (no password required), would then attempt passwords and measure response timings, which will leak the number of correct bytes in the password.

Level of difficulty: Low  
Chances of success: Medium  
Usual countermeasure: Easy, check passwords in constant time

MSP430 Bootloader: Method 3  As the password is stored in IVT, the availability of firmware image of the device contains the password, of course for this attack the attacker has to be a local adversary. An attacker with the access to the source code of the program can obtain the password by recompiling the program using the same compiler and options, which would lead to the same firmware image [3].

Level of difficulty: Low  
Chances of success: Medium  
Usual countermeasure: Easy, protect source code  
Secondary countermeasures: Easy, randomize IVT

MSP430 Bootloader: Method 4  In [34] a Half-Blind Attack on the MSP430 is presented. The MSP430 has 16 general purpose registers. The idea is that if the bootloader is locked and the firmware image is unknown, abusing a memory corruption can be used to directly call the known bootloader, however some sequences of instructions needs to be chained together but their locations are unknown. Usually during hard entry (from a reset) to the bootstrap loader, there are three instructions that are executed. The first instruction sets the stack pointer to address 0x0220, a valid SRAM position commonly used for the stack. The second instruction clears a register (R11). The password checking code sets bit 4 of R11 after proper password authentication. Clearing this bit during hard entry to the bootstrap loader ensures that a password must be presented before access. Where as these three instructions are not run in soft entry point (entry from local software). In [34] it is proposed to inject the soft entry address, 0x0c0e, into the program counter by overwriting the return pointer of the stack. Which provides access to the BSL without first clearing r11, such that if bit 4 of is set, access is granted to the BSL as if a valid password had been entered. Later the attacker could use the TX Data Block command to read the password.

Level of difficulty: Medium  
Chances of success: Medium  
Usual countermeasure: Medium, Address Space Randomization  
Secondary countermeasure: Medium, do not map Bootloader in normal address space

Memory Persistence Attacks  In microcontrollers, SRAM memory is used as processing memory, it therefore contains all temporary variables and registers values, key material is usually present in flash memory as well as in SRAM memory for processing, sometimes in a different state (round keys). On locked microcontrollers

\[\text{below 1.60 or above 2.12}\]
Table 2: Registers usage, omitted registers are general purpose

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>const zero</td>
</tr>
<tr>
<td>R1</td>
<td>stack pointer</td>
</tr>
<tr>
<td>R2</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>R3</td>
<td>Return value</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>R9</td>
<td>Link Register</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

none of those memories are accessible to the attacker. Microcontrollers usually allows a user to unlock the microcontroller but at the same time the Flash memory will be erased and the microcontroller will be reset. While the key material is not available anymore from the Flash memory it may remain in SRAM if this memory is not erased from SRAM. This is similar to cold boot attacks\[36\]. This attack was performed on an ember device by Travis Goodspeed to recover Zigbee AES key. Specifically, after sending an unlock/erase flash command a new program is loaded on the device flash memory, this program dumps the whole SRAM memory to a serial port, the attacker then analyze this data and it is very likely that the key will be present in this memory dump, as long as it was used before the erase/unlock command was sent.

**Level of difficulty:** Low  
**Chances of success:** High  
**Usual countermeasure:** Low, Erase SRAM on unlock or flash erase command

### 4.4 Evaluation of attacks on Coalesenses and IHP nodes

**Conditions of the evaluation**  
The chip from Jennic is a custom ASIC, that embeds a core, radio and many peripherals, while it is derived from an open source design it has significantly deviated from it. Jennic provides a limited amount of low level information on the chip. For example the datasheets are incomplete, not covering low level details (instruction set, general purpose registers, configuration registers, etc.). The GCC based compiler source is not publicly made available, it is only available on request. Furthermore, the iSense device we evaluated did not provide access to the JTAG pins, preventing the use of a low level debugger. Finally, an instruction set emulator is not available. While those limitations would not prevent a determined attacker from obtaining such information, they made this evaluation harder. This however should not be considered as a security feature. This information could be obtained by several means, e.g., from the easiest to the hardest: registering as a developer, requesting GPL licensed source code, buying a license to a related processor, reverse engineering from partial datasheets, compilers and assemblers [2], etc. Because of the limited time to perform this study, we provide an evaluation from the available information and limited reverse engineering. Among interesting features of the Jennic chip it has a 32bit address space in which are mapped 128KB of ROM code and 128KBytes of RAM memory [29]. The ROM memory (Figure 29(b)) contains a bootloader and a low level API code (802.15.4 mac layer, device drivers...). The RAM memory (Figure 29(c)) holds the program (loaded from an external flash), as well as dynamic data (heap, stack...). While some instructions are standard and obvious to understand (b.addi, b.or ...) some have a more unclear behavior (b.enti <byte><byte>; b.rtnei 0x5,0x0; ...), comments in the GCC source code was also helpful to better understand stack layout and register usage (Figures 31-30 in appendix). We report in Table 2 the usage of main general purpose registers.

**Stack-Based Buffer Overflow and Return Address Overwrite**  
As seen previously Stack-based buffer overflows is the most common software attack. We therefore developed and validated the feasibility of a stack-based buffer overflow with a sample application (Figure 32). This application emulate a real vulnerability by performing a memcopy to a too small stack allocated buffer. The return address is overwritten and changed so that the function will return a few bytes after its normal return location. In this example this change of return address will skip a few instructions that set the return value of the parent function. The resulting execution log that includes the relevant stack trace can be seen in Figure 33.

**Level of difficulty:** Low  
**Time (incl Reverse engineering):** Days
Code Execution on the Stack  Injecting code in the stack and executing it is the simplest way of injecting code because it can be done using the same vulnerability that was used for the stack based buffer overflow. We evaluated if stack memory was executable by initializing a stack allocated array with a few bytes that decodes as valid instructions. This code only resets a register and returns. We then included assembly code in the function to execute this code, after its execution we verify that the value of the register was reset, in a second test we jump after the instruction that resets the register, this proves that the code on the stack was actually executed. The code and execution logs are reported in Figure 34.

Level of difficulty: Low

Time (incl Reverse engineering): Days

Code Borrowing from ROM  Sometimes it is difficult to return to the correct address in the stack, or specific countermeasures are in place to randomize it. In most cases it is easier to return to a well known and fixed address where code is present. Because the attacker may not know the exact application code in memory in advance relying on the code present in the ROM memory, at a fixed address, is helpful for an attacker. We therefore evaluated if it is possible to return execute chunks of code from the ROM memory directly from the application code. For this we first used a custom program to dump ROM contents to the serial port, which we disassembled using the objdump tool provided in the BA2 toolchain. Once we had this disassembled ROM memory we located an address that contained a sequence of instructions that, similarly to the previous example, resets register R3 and returns (through a jump to the link register):

41f5: 00 60 b.movi r3,0x0
41f7: 47 d2 48 b.jr r9

Knowing this address, as done previously we called it from assembler code and verified that the code was correctly executed. The resulting test code is reported in Figure 35.

Level of difficulty: Low

Time (incl Reverse engineering/ROM dumping): Days
<table>
<thead>
<tr>
<th>Type of instruction</th>
<th>Alleged instruction behavior</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.reti 0x2,0x0</td>
<td>Return and free local variables on stack</td>
<td>437</td>
</tr>
<tr>
<td>b.jalr r7</td>
<td>Jump and link to register</td>
<td>189</td>
</tr>
<tr>
<td>b.jr r2</td>
<td>Jump to address in register</td>
<td>492</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>1118</strong></td>
</tr>
</tbody>
</table>

Table 3: Potential control-flow high jacking Gadgets in the ROM memory of JN5148. e.g. to be used for return oriented programming.

**Return Oriented Programming**  
Return oriented programming is among the most powerful attack techniques currently existing. It’s main advantage is that it can be used to perform arbitrary computation, without code injection. The feasibility of return oriented programming depends on a number of factors, among them the number of potential gadgets, sequences of instructions that terminate with a branch that may be controlled by the attacker (by controlling the value of a register or the stack). In Table 3 we report the number of possible gadgets (does not consider unintended returns [59]) that are present in the ROM memory of the Jennic JN5148. To summarize most of the requirements for the feasibility of ROP are present on the Jennic device. Among them:

- Lack of address space randomization.
- Presence of a large amount of fixed code at a known location.
- Possibility of manipulating the stack contents, modify stack pointer.
- Intensive usage of indirect jumps (To be used in return oriented programming without returns [17]).
- Large number of returns.
- Non constant sized instructions, this allows to find unintended gadgets, i.e. starting instruction decoding in the middle of the original instruction[59].

However, developing a return oriented programming framework would require:

- A better understanding of the instruction set
- Better development tools (debugger or emulator)
- More reverse engineering for the ROM code, and developing tools for that (e.g. IDA Pro plugin[10])
- Design or adapt a compiler to compile programs into “return oriented programs” (e.g. as done in [54]).

While performing the above steps are out of the scope of this study, a motivated attacker is likely to succeed.

**Level of difficulty**: Hard  
**Success Likelihood**: Medium/high  
**Time (exp.)**: Months

**Stack Overflows**  
The datasheet of the Jennic JN5148 contains a note about a stack overflow protection, however little details are provided. Stack overflows can be caused by 2 main factors:

- attacker controlled stack allocations
- recursive function calls
- large allocations on the stack

Variable sized stack allocation is not working on the iSense device, for example the following C99 compliant code compiles but hangs without error messages when executed:
In the following code example we test the second most common

```c
int ISenseDemoRet::test_stack_overflow_set_sp_2( int count ) {
    uint32 old_sp;
    uint32 new_sp_good=0x401f300;
    uint32 new_sp_bad =0x401d300;

    asm ("b.mov %0, r1 \n
         : =r"(old_sp) ::);
    os_debug("old sp = %x ",old_sp);

    // setreg("r1", new_sp_good); // within stack space should not "break too much"
    setreg("r1", new_sp_bad);  // within stack space should not "break too much"
    getreg("r1");
    toggle_led();

    // restore the stack
    setreg("r1",old_sp);

    return 1;
}
```

The above code works without problems, while the stack region is expected to be only 0X1000 bytes starting at address 0x401F000\(^7\). While providing an address below the normal limit of the stack (0x401D300) did not modify the behavior, providing an address in another memory section actually lead to a silent crash (The memory region was specifically reserved in the “text” region so that it was corruptible without crashing the device because of the corruption itself). In addition it appears that setting the stack pointer value to an incorrect value does not lead to problems, reading the value of the stack pointer when its value is incorrect leads to a silent crash. For example the following code does not creates errors:

```c
uint32 old_sp;

asm ("b.mov %0, r1 \n
     : =r"(old_sp) ::);
os_debug("old sp = %x ",old_sp);
os_debug("before ");
asm (" b.addi r1,r0,0x100 " \n
     // set SP
     " b.mov r1,%0 " \n
     :: "r" (old_sp):
     );
    os_debug("after ");
```

While the following code that reads the stack pointer value leads to a silent crash:

```c
uint32 old_sp_sp_val;

asm ("b.mov %0, r1 \n
     : =r"(old_sp) ::);
os_debug("old sp = %x ",old_sp);
os_debug("before ");
asm (" b.addi r1,r0,0x100 " \n
     // set SP
     " b.mov %0,r1 " \n
     // read SP <= crash here ?
     " b.mov r1,%0 " \n
     :="r"(sp_val):
     "r" (old_sp):
```

\(^7\)According to linker scripts and symbols in elf files.
### Device

<table>
<thead>
<tr>
<th>Attacks</th>
<th>MSP430</th>
<th>OpenMSP430</th>
<th>IHP430</th>
<th>Jennic JN5148</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer overflow/ret address corruption</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Executable stacks</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Ret To libc</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>ROP/borrowing from ROM code</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>memory persistence</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>memory persistence on unlocking device</td>
<td>+</td>
<td>+</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

### Countermeasures

| W⊕X presence                         | -      | ++         | -?     | -             |
| stack overflow protection            | -      | -          | -?     | +             |
| JTAG deactivation                    | +      | +          | +?     | +?            |

Table 4: Summary of the system level attacks, the difficulty to implement them and the likelihood of success. Legend: - impossible, + possible (under some conditions), ++ possible (verified), ? lack of data, +? likely, -? unlikely.

```c
os_debug("sp value =\%x\",sp_val);

os_debug("after ");
```

While we have seen some “Stack overflow” error messages on the iShell console they were most often due to mis-aligned memory accesses, and should not be reported as stack overflows. Because of the lack of availability of detailed documentation it is difficult to accurately determine the status of the stack overflow protection. However, from the experimentation above it seems that the stack overflow protection is enabled, but the limit of it are not well known.

**Level of difficulty:** Low/Medium  
**Success Likelihood:** Very Low  
**Time (exp.):** Days

#### 4.4.1 IHP Nodes

Unfortunately we were not able to evaluate the MSP430 based devices (IHP430), because of export regulation (Switzerland, where the partner involved in this study is, not being part of EU). Because of this limited availability of the devices we could not evaluate the actual devices. However, those devices are binary compatible with the original MSP430 and IHP reported to us that no specific countermeasure against software attacks were implemented. Therefore, the devices from IHP should be vulnerable to the attacks described above (Section 4.3). However, it is interesting to note that data memory is not executable on some implementations of the MSP430 (OpenMSP430).
5 Conclusion

The black box attacks on the Jennic JN5148 wireless microcontroller and on the IHP sensor node based on a Leon2 processor were successfully performed using a straightforward correlation-analysis attack approach. We stuck to only a few basic assumptions a black box attacker would also have at his disposal: (1) the power consumption / the EM emanation of a CMOS circuit is related to the Hamming Weight of the processed values in a circuit and (2) the nonlinear S-Box transformation represents a suitable point to mount a side-channel attack on an arbitrary AES implementation. This way we have shown that both sensor nodes are vulnerable to power/EM analysis attacks to a certain degree, even in case of using only very basic analysis methods. In Section 3.3.1 we demonstrated that an attacker provided with some additional information about the attacked device could reveal the secret key with a significantly lower effort.

In this document we report the state of the art on software attacks on general purpose computers, we then reviewed software attack techniques and countermeasures that are applicable to low end microcontrollers. We then evaluated the potential exposure of the current devices of interest to those attacks and their possible available countermeasures. For this we rely on previously available data and we made an extensive analysis of the Jennic JN4158 used by Coalesenses.
References

[1] LeCroy Active Differential Probe AP034.


[19] Coalescenses. isense core module datasheet cm20i, cm20u, cm20hp. [http://www.coalescenses.com/download/data_sheets/DS_CM20X_1v0.pdf](http://www.coalescenses.com/download/data_sheets/DS_CM20X_1v0.pdf).


A Validation of Software Attacks on BA2 Processor

/* [...] On the ba, these are r0 as 
a hard zero, r1 as a stack pointer and faked cc, arg and frame 
pointer registers. 
[...] */

[...] /* List the order in which to allocate registers. Each register must be 
listed once, even those in FIXED_REGISTERS. 

We allocate in the following order: 
r7-r4 (not saved; highest used first to make less conflict) 
r23-r31 (not saved) 
r2 (not saved; would use r3 for DImode/DFmode) 
r3 (not saved; return value register) 
r8 (not saved, would use r9 for DImode/DFmode) 
r9-r22 (saved) 
mac 
r0, r1, flag, arg, frame (fixed) 

This allocation order is optimal for ABI level 3 (default). */

Figure 30: BA2 registers usage, (from BA-GCC source comments)

/* Stack frame layout we use for pushing and poping saved registers: 

+----------------------+
| Ret addr | \ 
| (Previous FP) | | 
| Saved registers | | Previous stack frame 
| Local variables | | 
| Called function args | / 
| ....................+ 
| Saved varargs | <- adjusted by current func 

r2--> +----------------------+
| Ret addr | \ 
| (Previous FP) | | 
| Saved registers | | Current stack frame 
| Local variables | | 
| Called function args | / 

r1--> +.......................+

*/

Figure 31: BA2 stack layout, (from BA-GCC source comments)
uint32 shellcode[]={
    // 4 memory words
    0,1,2,3,
    0, // overflow will start here, add padding
    0xdeadaeef // overwriting the return address
};

void iSenseDemoRet::test_stack_based_buffer_overflow_sub( void )
{
    uint32 array[4];
    for( int i=0; i<4;i++){
        array[i]=0x0a0a0a0a; // eyecandy pattern
    }
    shellcode[5]=(uint32)tmp; // update "shellcode" with an arbitrary return address
    getreg("r1"); // prints stack pointer
    showmem(0x401FF28, 0x401FF90); // prints region of the stack frame
    memcpy(&array, &shellcode, sizeof(shellcode)); // stack based buffer overflow
    os_.debug("after memcpy");
    showmem(0x401FF28, 0x401FF90); // prints region of the stack frame after overflow
}

int iSenseDemoRet::test_stack_based_buffer_overflow( void )
{
    volatile uint32 result=0;
    // get modified address to return to in a global variable
    tmp=&&set_result;
    os_.debug("before call, address of faked return %x",tmp);
    test_stack_based_buffer_overflow_sub();
    // <= return address normally points here
    result=1; // result variable set to 1
    // if return address changed to this address "result" will remain == 0
    set_result:
    return result; // e.g. password wrong
}

void iSenseDemoRet::execute( void* userdata )
{
    int res=test_stack_based_buffer_overflow();
    if(res==1)
        os_.debug("Ret address overwrite fail.");
    else
        os_.debug("Ret address overwrite Success!!");
}

Figure 32: Example stack based buffer overflow with modification of return address used by test_stack_based_buffer_overflow_sub.
Figure 33: Execution log for stack based buffer overflow validation example.
void iSenseDemoRet::test_exec_stack( void )
{
    volatile uint32 result;
    volatile uint8 array[]={0x00, 0x60, 0x47, 0xd2, 0x48};
    // instructions to be executed from stack
    // 00 60 b.movi r3,0x0
    // 47 d2 48 b. jr r9
    uint32 jump=(uint32)&array;
    asm (" b.addi r3,r0,0x100 " // set r3 to 0x100
          " b.mov r10,0x0 " // set r10 to address to jump to
          " b.jalr r10 " // jump and link to gadget
          " b.mov %0,r3 " // " \t\n" // read value of r3 after gadget execution
           :"=r" (result)
           :"r" (jump) // this is the address we want to jump to ...
           :"r3","r9","r10"
        );
    os_.debug("after jump to stack, r3=%x",result);

    asm (" b.addi r3,r0,0x100 " // set r3 to 0x100
          " b.mov r10,0x0 " // set r10 to address to jump to
          " b.jalr r10 " // jump and link to gadget
          " b.mov %0,r3 " // " \t\n" // read value of r3 after gadget execution
           :"=r" (result)
           :"r" (jump+2)
           :"r3","r9","r10"
        );
    os_.debug("after jump to stack (no r3<=0) , r3=%x",result);
}

// Execution Log
// [16:41:44.800] App::boot
// [16:41:45.812] after jump to stack, r3=0x0
// [16:41:45.832] after jump to stack (no r3<=0) , r3=0x100

Figure 34: Test for validation of executable stacks. The code present is in the (stack allocated) variable “array” is erasing register R3 and returns, the first call to it is made with R3 set to 0x100, value of R3 is printed after execution, if the code in stack is executed properly R3 will be set to 0.
```c
void  iSenseDemoRet::test_ret_rom( void )
{
    volatile  uint32 result;
    uint32 gadget_addr=0x41f5;  // address of a b.jr r9
    // example gadget in ROM that sets r3 to zero and returns to address in link register
    // 41f5:   00 60  b.movi r3,0x0
    // 41f7:   47 d2 48  b. jr r9

    // Step 1
    asm ( " b.addi r3,r0,0x100 " "\t\n"  // set r3 to 0x100
       " b.mov r10,%0 "  "\t\n"  // set r10 to address to jump to
       " b.jalr r10 "  "\t\n"  // jump and link to gadget
       " b.mov %0,r3 "  "\t\n"  // read value of r3 after gadget execution
          ":=\t" (result)
          :
          "r" (gadget_addr)
          :
          "r3","r10"
    :
    );
    os_.debug("after jump to rom, r3=%x",result);

    // Step 2
gadget_addr +=2;  // now do the same but skip the "b.movi r2, 0x0" instruction
asm ( " b.addi r3,r0,0x100 " "\t\n"  // set r3 to 0x100
       " b.mov r10,%0 "  "\t\n"  // set r10 to address to jump to
       " b.jalr r10 "  "\t\n"  // jump and link to gadget
       " b.mov %0,r3 "  "\t\n"  // read value of r3 after gadget execution
          ":=\t" (result)
          :
          "r" (gadget_addr)
          :
          "r3","r10"
    :
    );
    os_.debug("after jump to rom (no r3<=0) , r3=%x",result);
}
```

Figure 35: Validation of the lack of protection against arbitrary execution of pieces of code from ROM on the JN5148. In the fist block we set up register 3 to the value 0x100, jump to a piece of code that reset it and returns (through Link register) the log shows the r3 value reset. The second part of the code jumps to the return instruction only (gadget_addr is increased), as expected the value of r3 is not reset anymore. This validates the correct execution of arbitrary code from the ROM memory, opening the possibility to develop a gadget set from the ROM code.