Intrinsic Code Attestation by Instruction Chaining for Embedded Devices

Oliver Stecklina[1](✉), Peter Langendörfer[1], Frank Vater[1], Thorsten Kranz[2], and Gregor Leander[2]

[1] IHP, Im Technologiepark 25, 15236 Frankfurt (oder), Germany
{stecklina,langend,vater}@ihp-microelectronics.com
[2] Horst Görtz Institute for IT-Security (HGI), Ruhr-University Bochum, Bochum, Germany
{thorsten.kranz,gregor.leander}@rub.de

Abstract. In this paper we present a novel approach to ensure that no malicious code can be executed on resource constraint devices such as sensor nodes or embedded devices. The core idea is to encrypt the code and to decrypt it after reading it from the memory. Thus, if the code is not encrypted with the correct key it cannot be executed due the incorrect result of the decryption operation. A side effect of this is that the code is protected from being copied. In addition we propose to bind instructions to their predecessors by cryptographic approaches. This helps us to prevent attacks that reorder authorized code such as return-oriented programming attacks. We present a thorough security analysis of our approach as well as simulation results that prove the feasibility of our approach. The performance penalty as well as the area penalty depend mainly on the cipher algorithm used. The former can be as small as a single clock cycle if Prince a latency optimized block cipher is used, while the area overhead is 45 per cent for a commodity micro controller unit (MCU).

1 Introduction

Embedded devices especially when used in automation systems are becoming more and more often target of attacks. The modification of embedded systems software is extremely dangerous. Especially in cyber-physical systems (CPSs) such as energy distribution networks any penetration and modification can cause disasters. Common approaches cannot ensure that an embedded system runs the code that was initially deployed. Code injection attacks are feasible on any architecture. By using return-oriented programming (ROP) attacks [33] code can be injected even on Harvard architectures as shown in [17].

In order to prevent successful attacks and to detect alteration of the code deployed on the embedded devices quite some approaches have been researched in the last few years SWATT [32], SMART [15], etc. All these approaches share

The research leading to these results has received funding from the Federal Ministry of Education and Research (BMBF) under grant agreement No. 16 KIS0004.

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2015
B. Thuraisingham et al. (Eds.): SecureComm 2015, LNICST 164, pp. 97–115, 2015.
DOI: 10.1007/978-3-319-28865-9_6

a common drawback. They check whether the code originally deployed was changed or whether additional code was injected. Even if they work 100 per cent correct they cannot prevent malicious code from being executed, nor can they prevent ROP attacks. In this paper we present an approach we call intrinsic code attestation. The core idea is to execute encrypted instructions, so only instructions that are authorized can be executed. Consequently, no malicious code can be inserted. In addition we "chain" instructions so that a certain instruction can be executed only after its predecessor. This prevents ROP based attacks. As an important side effect enciphered code to be deployed on the embedded devices protects the code from being stolen by an adversary. We denote our approach as intrinsic code attestation (ICA). The main contributions of this paper are:

– Introduction of core principles of ICA, especially how chaining of instructions can be ensured for non-sequential program flows e.g. if jump instructions or branches are used.
– Discussion of simulation results that show on the one hand that our approach can be implemented with existing widely used micro controller unit (MCU) architectures and on the other hand that the performance penalty is a single clock cycle only.
– Thorough security analysis of the ICA approach including the discussion of collisions of the nonce used for instruction chaining in ICA and brute forcing encrypted instructions.

The rest of this paper is structured as follows. Section 2 details the ICA concept. Our security analysis is presented in section 3. The following section provides the implementation of ICA in an MSP430 simulation environment and for a 8-bit VLIW RISC processor. Related work is discussed in section 5, while section 6 and 7 present future work and conclusions, respectively.

2 Intrinsic Code Attestation

The core idea of intrinsic code attestation (ICA) is to ensure that only authorized instructions can be executed on a certain MCU and that also their sequence is fixed. The presented approach is based on a standard block cipher to provide a high security level. We use the block cipher in the counter mode (CTR) to overcome the block size limitation when encrypting sole instructions. The block cipher is parametrized by an individual program key (IPK) and an instruction individual key (IIK). The IPK guarantees that the program text cannot be read by an adversary to gather intellectual property (IP). The IIK is used to built an instruction chaining that ensures that instructions cannot be reordered or invoked from extrinsic program locations.

2.1 Instruction Chaining

Figure 1 illustrates the idea of a crypto-based instruction chaining. Information of instruction (n) are input of a cipher that decrypts instruction $(n+1)$. In case of a manipulation of the program flow any out of order instruction is decrypted with wrong cipher inputs, which results in an illegal or at least an unpredictable instruction. Since an instruction chaining by using the instruction as input for the cipher strictly binds an instruction to its previous instruction, non-sequential program flows become infeasible. Due to such a restriction cannot be applied to real applications our chaining is based on additional information. Hence, we extended each instruction by an individual *nonce*, the IIK, that is encrypted in conjunction with the instruction. The nonce is used as input for the cipher to decrypt the succeeding instruction. Using individual nonces prevent a modification of the program flow similar to applying the instruction to cipher. However, in addition non-sequential program flows can be encrypted as well.

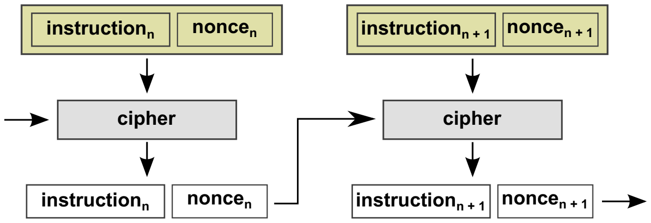

Fig. 1. An ICA can be enforced by a crypto-based instruction chaining so that an instruction cannot be decrypted without executing the previous one.

An insuperable program code encryption can only be guaranteed if the decryption unit is integrated in the processor's data path without any bypass. Hereby, each instruction must pass the decryption unit before its execution. Wrong key information will result in illegal or unpredictable instructions, which are passed to the instruction decoder and cause an illegal instruction trap or an unpredictable behavior. Therefore, the IPK and the IIK must be stored in a secure manner. The IPK storage will be illustrated in Section 4.3. The IIK is decrypted with an instruction and hold inside the decryption unit for decrypting the succeeding instruction. Any external access to the key is unnecessary and may not be implemented.

Conditional Jumps. A non-sequential program flow is generated by each conditional jump. As shown in Figure 2, a jump instruction has two possible successors. Due to dynamic program flow both predecessor instructions must be considered. Therefore, two identical IIKs are used to encrypt the jump instruction ($instrA$) and the instruction immediately before the jump target ($instrC$).

But by using two identical IIKs for one instruction a program flow modification becomes possible. It cannot be guaranteed that the program does not jump from instruction $instrC$ to $instrB$. The remaining risk of such a modification is analyzed in Section 3.

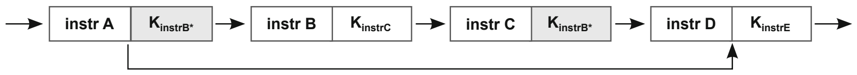

Fig. 2. Conditional jumps require that both possible jump target (*instrB* and *instrD*) are encrypted with the same IIK (K_{instrB}).

Function Calls. Beside conditional jumps each function call generates a non-sequential code sequence as well. Figure 3 illustrates a call of a function by two different threads. Each caller attaches the nonce of the first callee instruction to its call instruction. This ensures that the considered function can only be called. Furthermore, each instruction just behind the call instruction must be encrypted by the nonce that is attached to the return of the callee. Although, this enforces that a return instruction cannot be used to jump to any instruction, as is used by ROP attacks, an attacker can modify the program flow to jump to any thread that calls the function. Although the instruction chaining reduces the attack vector significantly a remaining risk is still there.

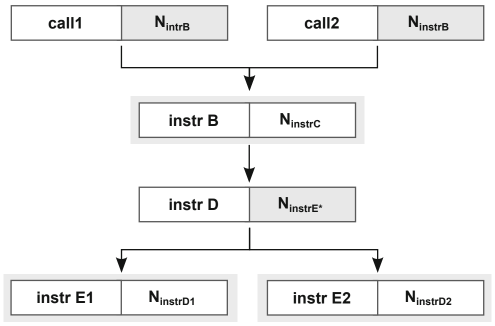

Fig. 3. Callers must attach the same nonce to the call instruction and instructions just after the call must be encrypted by a nonce attached to the return of the callee.

Strict binding of a callee to a caller makes dynamic function calls impossible. Therefore, function pointers and polymorphism cannot be used with ICA. However, this restriction can be mostly circumvented by using trampoline functions.

Asynchronous Events. On real processor the program execution flow can be interrupted by an asynchronous event. Such an event is a signal from a peripheral unit or an internal exception that needs immediate attention. Software includes service routines to deal with event. The interruption is temporary, the processor resumes to normal activity after finishing the service routine.

The ICA approach has to deal with asynchronous events to be suitable for real world applications. Due to that the asynchronous events can interrupt any instruction the nonce must be provided externally and the current nonce must be saved while handling the event. In case that nested events are allowed a nonce stack to store the current nonces is necessary. However, the maximum stack size is equal to the number of interrupts, which is usually small on embedded systems.

2.2 Instruction Key Expanding

Each instruction is encrypted with an individual nonce. Due to the fact that the suffix inflates the program size a minimal nonce must be chosen. But since the nonce is used as input of the block cipher it must be expanded to the size of the block cipher. In a simple way as shown in Figure 4 (a) the nonce can be padded to the block size with zeros.

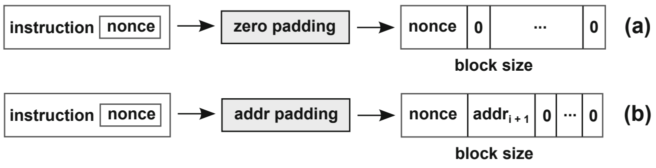

Fig. 4. IIK expanding by padding zeros to the nonce (a) or including the instruction address (b).

Since the nonce have no relation to the instruction address the enciphered program code can be used on any location. It can be circumvented by applying the instruction address to the cipher. Due to the decryption unit is integrated in the MCU's memory path the instruction address is available there. As shown in Figure 4 (b), the address can be appended to the nonce and only the remaining bits are padded by zeros. We discuss the advantage of such an instruction pinning in more detail in Section 3.

2.3 Instruction Size Fitting

Depending on the MCU's instruction set architecture (ISA) the block size of the chosen cipher does not need to be identical to the length of the instruction plus the nonce. Therefore, we use a symmetric block cipher in a CTR to generate a temporary instruction key (TIK) as shown in Figure 5. The XOR-operation uses the first n-bits of the TIK to decrypt the instruction and the nonce. Due to non-sequential code the counter is reseted with each instruction. Therefore, the TIK depends on the nonce and the address if used only.

If an instruction plus nonce is longer than a single cipher block the IIK is incremented to generate an additional TIK block. The cipher stream builds

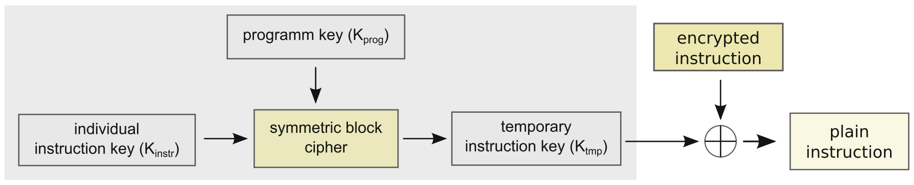
programm key (K_{prog})
encrypted instruction
individual instruction key (K_{instr})
symmetric block cipher
temporary instruction key (K_{tmp})
plain instruction

Fig. 5. Instructions are decrypted by using the CTR of a symmetric block cipher. The block cipher gets the IPK and a IIK to generate the TIK.

a TIK with a proper length. However, any additional block causes additional performance penalty, a block cipher should be chosen that has a suitable block size or encrypt speed.

3 Security Analysis

Due to it is difficult to quantify the security benefits of any given technology. The effects of unexploited vulnerabilities cannot be predicted and real-world attacks can be thwarted by trivial changes to those details. Therefore, our presented security argument is informal. A more substantial argument (or a proof) would require formal analysis and verification of the ICA hardware implementation. The security of the ICA approach is based on the following assertions:

A1 The TIK calculated by the memory decryption unit (MDU) cannot be forged. Since the TIK is the result of a strong block cipher with an adequate security level.
A2 The program key can be accessed only from within the MDU. This is guaranteed by the absence of physical lines to read the key outside.
A3 Physical and hardware-based attacks on the MDU are beyond the adversary's capabilities.
A4 The MDU cannot be bypassed since it decouples the instruction memory from the instruction decoder. All instructions must pass the MDU.
A5 The nonce cannot be replaced by a user defined value. The hardware guarantees that the nonce is directly read from the encrypted instruction memory.
A6 An instruction can be only decrypted with the correct nonce. The nonce and the instruction address are the initialization vector of the CTR block cipher.
A7 The program key update is forbidden or protected by a strong authentication scheme.
A8 Any erroneous decryption results in an unpredictable program behavior or leads to a hardware reset.
A9 The normal execution of an encrypted program should leak no information about the program key and the encrypted nonces.

Considering these assertions the system's security is mainly determined by the resources spent for the ICA implementation. Especially the nonce size and the ISA have a major impact on the remaining risk.

3.1 Remaining Risk

Due to our approach is mainly based on individual nonces, we focused our remaining risk analyze on attacks on the nonces as well as on the instruction chaining. We assume that a 16-bit nonce was chosen. It is a good compromise between minimal nonce size and memory overhead. In the following we discuss the effect of key collisions, brute-force attacks, and attacks based on ROP.

Instruction Key Collisions. In case of using all 16-bit nonces the number of TIKs is determined by the number of images of the block cipher function. When using the first 16-bit of a block cipher output with a block size larger than 16-bit the number of images is approximately $2^{16}(1 - e^{-1})$. Using the instruction address within that enlarges the input domain does not affect the number of images. Due to commodity 16-bit MCUs have an address space up to 22-bit TIK collisions cannot be avoided. On an architecture with 22-bit address space each TIK may be used up to 100 times.

However, from the perspective of security the reduced number of TIKs and their multiple used is harmless. Since an attacker does not know the IPK it cannot qualify the correct set of TIKs. The probability of guessing a precise nonce of an instruction remains 2^{-16}. Furthermore, in case of randomly spreading the nonces over all instructions the multiple use of a TIK does not increase the probability as well.

Cipher Instruction Search Attack. The idea behind a *cipher instruction search (CIS) attack* is presented by Kuhn [24]. It is based on a brute force attack on the enciphered machine instructions and then observing the CPU reaction. The adversary presents a large number of guessed encrypted machine instructions to the CPU to construct an enciphered program to gain more information or to provide cleartext access to the instruction memory.

For a CIS attack the target device must be connected to a programming device. We must assume that the device provides access to all processor registers except the MDU internal registers. Depending on the system architecture instruction memory may be non-volatile memory (flash) or RAM. Due to flash modifications are very complex and the low flash endurance, flash based attacks can be neglected. An architecture that executes instructions located in the RAM is much more vulnerable for CIS attacks. Depending on the speed of the programming device an attack can be done quite fast. At an MCU clock speed of 20 MHz a shot of a single instructions needs only few milliseconds. So brute forcing all 2^{16} alternatives takes only few seconds. Furthermore, the brute force strategy can applied to each instruction in the same way with the same effort. Hence, on a von-Neumann architecture with shared instruction and data memory, an enciphered program can be constructed in short time.

The success of a (CIS) attack can be significantly reduced on systems with larger instructions. Furthermore, adding the instruction address to the nonce pad prevents a copy of guessed program code and makes the reuse of an enciphered program code, which was constructed in a RAM, infeasible. Nevertheless, we assume that a gain or an update of the IPK is infeasible by guessing a program sequence if both are protected by additional schemes, which are not infected by that attack.

Multiple Return Points. The instruction chaining presented in Section 2 uses the nonce that was encrypted together with the previous instruction for decrypting the next instruction in a CTR wise fashion. Hence, for sequential code that does not include any branches, a unique nonce is used for encryption of each instruction. This uniqueness assures that only the legitimate previous instruction can be the predecessor of the current instruction. Any other instruction comes with a different nonce and will most likely propose a wrong TIK. That is because a good block cipher behaves similar to a random function. Since we use the first n bits of a block cipher output as the TIK, the probability that two given nonces propose the same TIK is approximately 2^{-n}. If a wrong TIK is used to decrypt an instruction this will result in an illegal or unpredictable instruction.

As soon as there are branches a nonce will be used multiple times, thus allowing an instruction to have multiple successors that can be decrypted with this nonce. This introduces the possibility of undesired modifications in the program flow: multiple instructions sharing a nonce are able to jump to each others successors. For example, in Figure 2 instruction *instrC* has a valid nonce for decrypting instruction *instrB*. Nevertheless, the number of instruction that might be jumped at is significantly reduced to the number of two.

A second risk occurs by legitimate jumps that might be taken when they are actually not allowed: the return instruction of a function might have multiple successors corresponding to multiple calling instructions. Although a jump to all these successors is legitimate in general, only one of these jumps should be taken at a certain point of time. Namely that one that returns to the instruction that was actually calling the function. The same obviously holds true for conditional jumps where both jumps are valid while only one of them should be taken at a certain point in time. Again, in both cases the attack vector is significantly reduced.

4 System Integration

To assess feasibility, practicality and impact of our approach we integrated it in the MSPsim. The MSPsim is a Java-based cycle accurate instruction set simulator (ISS) developed by the SICS [16]. It allows an execution of unmodified MSP430 firmwares. The ICA integration was done by implementing an additional Java module, which was bound to the instruction emulation module.

Beside the MSPsim extension we analyzed a more suitable tiny ISA and did deeper investigations on block ciphers and tool chains.

4.1 Secured MSP430

The MSP430 is a 16-bit MCU developed by Texas Instruments (TI). It uses a classical von-Neumann architecture with a shared data and program memory. The MCU is very popular in ultra low power applications and wireless sensor networks (WSNs). We used the MSP430 due to the availability of soft cores [20,28] and the MSPsim.

MSP430 Integration. All MSP430 instructions are structured in 16-bit words and the length of the instruction depends on the addressing mode. It differs from a single word up to four words. Therefore, the MSP430 instruction decoder performs multiple memory accesses within a single instruction. An integration of the MDU is shown in Figure 6. Each fetch is passed to the MDU and processed separately.

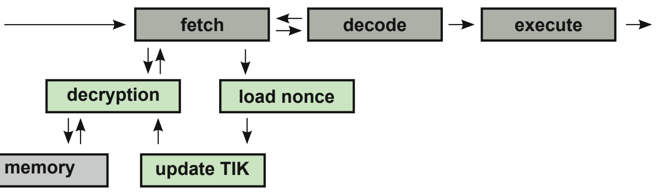

Fig. 6. The MSP430 performs multiple fetches within a single instruction, which must be separately decrypted. Finally the nonce is loaded and the TIK is updated.eps

The nonce is loaded automatically as an additional instruction word after loading all words of an instruction. We must only extend the instruction decoder to initiate the TIK update. The operation can be executed in parallel with the final instruction phases.

Instruction Encryption. Due to the variable instruction size the instruction encryption must be done in three steps. Listing 1 shows assembler text of a short loop program. The program starts at address 0x4000 and loops between the instructions at address 0x4006 and 0x4008. All instructions beside the move instruction use a single word.

```
00004000 <__ctors_end>:
   4000: 31 40 80 02  mov  #640,    r1
   4004: 02 43        clr  r2
00004006 <LOOP>:
   4006: 12 53        inc  r2
   4008: fe 3f        jmp  \$-2
```

Listing 1. Assembler program of a simple loop implemented on an MSP430.

The Listing 2 shows the instructions of Listing 1 extended by the nonces. We chose 16-bit nonces driven by the architecture of the MSP430. Since each nonce consumes two bytes of the address space the instruction addresses were changed. Hence, the jump instruction at address 0x400e had to be adapted accordingly.

```
00004000 <__ctors_end>:
  4000: 31 40 80 02 00 01
  4006: 02 43 00 02
0000400a <LOOP>:
  400a: 12 53 00 03
  400e: fd 3f 00 04
```

Listing 2. Listing 1 extended by the nonce. Due to the new instruction length the jump instruction had to be adapted.

In a final step the program is encrypted. Due to the CTR the size of the instruction has not to be changed and the encrypted instructions can be placed at their origin addresses.

Interrupt Handling. When an interrupt is requested from a peripheral the MSP430 executes at least the following: the currently executed instruction is completed, the program counter (PC) and the status register (SR) are pushed onto the stack, the SR is cleared, and the content of the interrupt vector is loaded into the PC. The next instruction continues with the interrupt service routine (ISR) at the given address.

The interrupt processing is extended by storing the current nonce inside the MDU and providing a predefined ISR nonce. Afterwards, the TIK can go on similar to the normal program execution. The ISR terminates with the instruction *reti*, which restores the PC, the SR and the instruction nonce. Since the MSP430 does not feature an *in interrupt flag* the ICA needs to be fully integrated in the interrupt logic to detect interrupts.

For each interrupt a static nonce is needed. The current implementation uses a single nonce for all interrupts. The nonce is stored inside the MDU and update is handled similar to the IPK.

4.2 Secured tinyVLIW8

The tinyVLIW8 is a size-optimized soft-core processor for deeply embedded control tasks [36]. We analyzed the tinyVLIW8 soft-core processor in addition to the MSP430 to evaluate the suitability of our approach on different system architectures. In contrast to the MSP430 the tinyVLIW8 features a Reduced Instruction Set Computer (RISC) architecture with a uniform instruction format and a Harvard architecture with a dedicated instruction, data, and IO memory. The processor executes two 16-bit instructions coded in a single 32-bit instruction word in parallel. Each instruction address points to a 32-bit instruction word. Hence, the nonce can be easily added by widening the instruction memory. Any adaptations to instruction address are not necessary.

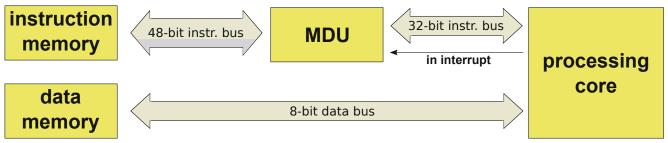

Fig. 7. The architecture of the tinyVLIW8 allows a placing of the MDU between the instruction memory and the processing core. Due to the dedicated data memory is not encrypted, it can be used unchanged.

Furthermore, due to the Harvard architecture a dedicated nonce load is not necessary. Instead the data bus can be split into a lower 32-bit bus for the processing core and additional n-bit bus for the nonce. Figure 7 shows a placement of the MDU. Only the instruction bus must be routed via the MDU. The data memory can be connected unchanged to the processing core. The interrupt handling of the tinyVLIW8 is similar to one of the MSP430. On an interrupt request the current instruction is completed and the PC is loaded from the interrupt vector table. But the processor provides an *in interrupt flag* that can be used to easily detect an interrupt service.

The fixed instruction size of the processor simplifies the integration of our approach in a significant manner. Furthermore, 32-bit instruction are much less vulnerable for CIS attacks. Without deeper investigations we are convinced that most of the RISC architectures with a uniform instruction length allow a similar integration. Possible candidates are the Leon2 or ARMv7 cores.

4.3 Secure Key Storage

The primary target of an attacker may be the program key of the device. Clearly, it cannot be stored in the systems memory, since malware code can easily access it and use it to encrypt additional malicious code. Therefore, the key is stored inside the MDU and readable from there only. However, installing new firmware images requires an export or import of the symmetric key. While a public key implementation could simplify the key management significantly, it comes with an unacceptable overhead. Hence, we propose three different approaches for management of a symmetric key: one-time programmable (OTP) memory, password protection, and physical unclonable function (PUF).

OTP Memory. An OTP memory is a memory where the setting of each bit is locked by a fuse or an antifuse. The memory can be written only once. It is possible after fabrication without any special equipment. Hence, the key can be set by the device owner before the first deployment. External read lines are not necessary, so that the key is only externally writable. But an OTP based key cannot be updated later. In case of a key revealing the device becomes insecure and must be replaced.

Password Protection. A secure IPK update can simplify the key management in case of a key revealing. Since the key is stored inside the MDU a memory mapped IO interface can be implemented to access the key. The interface can be protected by a password, which must be written in-front of the new key to unlock the memory mapped IO. A key read function is not necessary. Similar to the MSP430 boot-strap loader protection [38], the password can be stored within the firmware image. Because the firmware image is decrypted the password is protected by the current program key. A special protected memory is not necessary.

PUF. A very high security level can be provided by a local re-encryption of the firmware image. The firmware can be deployed with a shared program key, which can be stored inside the current firmware image similar to the password, described above. After deployment the image is re-encrypted with a device specific key on the device itself. Such a key can be based on a PUF, which provide true random numbers [21]. The PUF-based IPK must never leave the MDU. Such a re-encrypted firmware is bounded to the device, which prevents any firmware copy or off-line attacks on alternative devices.

4.4 Design Size and Speed Estimation

Due to using the block cipher in a CTR the instruction decryption works in two steps. First, the TIK must be generated by loading the nonce and running a block decryption. Second, the TIK is combined with the encrypted instruction by using an XOR-operation. As shown in Figure 8, the XOR-operation works in parallel with the data fetch and does not consume any additional clock cycles. However, the block cipher cannot be started before decoding the last instruction word and loading the nonce. Some cycles may run in parallel with block decryption, but the fetch of the next instruction must be delayed until the block decryption is finished. Hence, the instruction execution time is strongly coupled with the performance of block cipher.

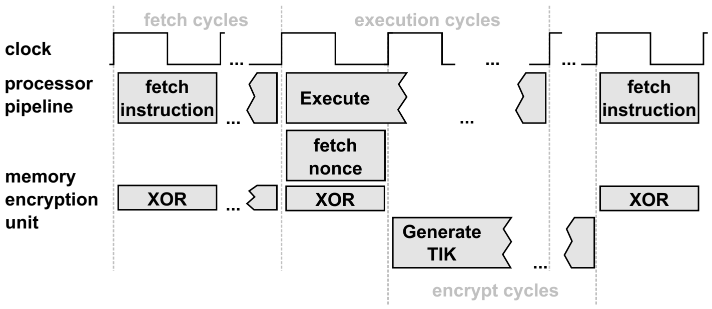

Fig. 8. The TIK can be generated in parallel with the execution phase of the processor's pipeline. But the fetch of the next instruction must be block until the TIK generation has been finished.

On the MSP430 few instruction are coded in four words. Hence, a 64-bit cipher needs two block operations for those instructions. Depending on the speed of the block cipher the instruction fetch may be interrupted twice. Therefore, on such a system the cipher speed becomes more significant.

We used the AES in the MSPsim extension. Therefore, we analyzed the AES algorithm as a candidate for a hardware implementation first. Table 1 gives an overview about the design size of the algorithm and the number of cycles for a single block decryption on a commodity FPGA. We analyzed a size-optimized and speed-optimized version. However, both were inadequate. The speed-optimized version need 12 clock cycles and is 8 times larger than our tinyVLIW8 processor (see Table 2). Therefore, we analyzed the PRINCE algorithm next [8]. The algorithm has a block size of 64 bits and is hardware-optimized. The size-optimized version needs 750 LCs only and is faster than the speed-optimized version of the AES. Furthermore, the PRINCE algorithm can be implemented in a fully unrolled version with a moderate design size enlargement. It needs 1.9 kLCs only and decrypts a block immediately. The maximum clock speed is limited by the longest logical path. In simulations we could measure a maximum end-to-end delay of 64,9 ns, which is equivalent to clock speed of 30.8 MHz.

Table 1. Design size of block ciphers (measured with Quartus II 11.0 Design Suite).

Cipher	Cycles	logical cells (LCs)	Regs	Fmax
AES [41]	60	2,403	428	53.7 MHz
AES [40]	12	8,855	792	104.9 MHz
PRINCE [8]	11	750	70	159.1 MHz
PRINCE (unrolled) [8]	0	1,875	0	30.8 MHz

Due to the power consumption is tightly coupled with the design size of the MCU the ICA extension must be based on a tiny cipher implementation. To evaluate the size impact of the block cipher, we analyzed soft-core MCUs on an FPGA. Table 2 gives an overview about the results. The larges MCU has a size less than 10 kLC and features a SPARC V8 ISA. The commodity MSP430 needs 2.8 kLC with a 16-bit ISA and 4.1 kLC with a 20-bit ISA. The size-optimized tinyVLIW8 soft-core needs around 1 kLC only.

We chose the tinyVLIW8 soft-core and the unrolled PRINCE version for a prototype implementation. Table 3 shows the sizes of the processor entities. The MDU entity includes the PRINCE implementation, the CTR, interrupt handling and the instruction decryption. Is quite smaller than the sole one, which is reasoned by absence of the external FPGA pins. Furthermore, we integrated a memory-mapped IO interface for a program key update. We can see, that the MDU overhead is 156 per cent for the tinyVLIW8 processor. A similar implementation for an MSP430 will result in an overhead of just about 44 per cent.

[1] The Leon2 design size could not be measured on Cyclone II, it was taken from [1].

Table 2. Design size of soft-core MCUs (measured with Quartus II 11.0 Design Suite).

Soft-core	LCs	FPGA
Leon2 [1]	9,299	Altera Cyclone[1].
openMSP [20]	2,841	Altera Cyclone II
IHP430X [28]	4,107	Altera Cyclone II
TinyVLIW8 [36]	1,162	Altera Cyclone II

The overhead is mainly driven by the cipher design size. The MDU, without cipher, needs only 126 LCs and 165 registers.

Table 3. Design size of tinyVLIW8 processor extended by the ICA approach (measured with Quartus II 11.0 Design Suite).

Entity	Design	Core	MDU	Peripherals	Dbg.-Inf.
LCs	2979	818	1817	234	110
Registers	558	224	165	113	56

4.5 Compiler Tool Chain Extension

The generation of an encrypted firmware is split in two steps. First, all instructions of the firmware are extended by the nonce. For this purpose the firmware must be analyzed to identify non-sequential instruction sequences. On an MSP430 furthermore, all jumps and calls must be adapted to new addresses. In a second step the instructions are encrypted.

The program analysis to identify non-sequential code can be done on the final firmware image or as an integrated step of the build process. Depending on the software and ICA the first approach could be complex. Therefore, we analyzed common build chains to identify possible candidates for an extension. The software of an MSP430 can be build with the GNU as well as the TI compiler. But both are not designed to be easy to extend. A more promising approach is to extent a modular build chain as provided by the LLVM project [25]. The LLVM tool chain splits the build process in a front-end step, an unrestricted number of optimization steps, and a back-end step. The two steps of the ICA encryption can be integrated as replacement of the origin back-end step.

A similar approach is provided by the CoMet tool [39]. CoMet uses any front-end compiler and can transform any intermediate code. In contrast to LLVM, based on intermediate codes a program can be simulated with the integrated simulator. Due to its flexibility and its simulation capability we decided to use it to generate the encrypted firmware.

5 Related Work

Approaches for a secure boot strap architectures to verify the program start [4] and stack protection schemes to prevent program flow modifications [18,34] work locally as well as with foresight, but leak a dynamic verification of the program code. An enforcement that a software follows a path determined ahead of time is provide in the work of Abadi et al. [2,3]. The control flow integrity (CFI) approach shares many ideas with methods that attempt to discern program execution deviation from a prescribed static control flow graph (CFG) [27,31,42]. While these works are focused on fault-tolerance, the CFI approached concerns with a persistent adversary that is able to change data memory, e.g. by exploiting program vulnerabilities. It ensures that an attacker can never execute instructions outside the legal CFG. But CFI inserts inlined labels and checks, which requires a program code modifications at run-time and does not provide any program code integrity. Furthermore, we are convinced that a secure approach must provide a dynamic program flow as well as a program code verification, which can be provided by none of these approaches.

Device attestation is the process of verifying the local state of a device. Previously proposed attestation techniques are mainly based on remote attestation protocols, where an external prover is used to verify the internal state of a device. These approaches can be differentiated in software-based [13,22,29,32], locally assisted by specialized secure hardware [15,30,35], and cluster-based protocols [23]. Though, the authors of software-based techniques argue that locally assisted approaches require specialized hardware, these approaches have been subject to successful attacks [10] and provide thus a disputable security level. Hardware-based approaches, such those based on local read-only memory [15,30], provide a secure anchor, which helps to overcome basic drawbacks of the software approaches. Beside binary attestation property-based attestation protocols are proposed [11,22]. These protocols are also assisted by specialized hardware and allow a blind verification and revocation of mappings between properties and configurations. Nevertheless, all these approaches work after the fact. If an adversary has successfully injected malicious code the victim operates out of its specification until a remote attestation detects the misbehavior.

Program code integrity and confidentiality is key in digital rights management and smartcard systems. Specialized processors with an integrated MDU are already state of the art. The DS5002FP and DS5240 secure microprocessors presented by Best [5–7] provide an execution of enciphered firmwares. But reasoned by the weakness of the used cipher the system can be broken by a CIS attack [24]. A security enhanced MMU (SMU) based on TDES is presented by Gilmont et al. [19]. In contrast to Best the approach uses the TDES cipher to encrypt the instruction memory. The work of Elbaz et al. gives an overview about hardware-based memory-bus encryption techniques [14]. It illustrates and compares the patent of Candelore [9], the SMU [19], the Xom approach of Lie et al. [26], and the AEGIS project [37]. The work of Chen et al. [12] presents a software-based approach, which uses a supervisor instance to decrypt instructions. But all these approaches are focused on memory encrypted to prevent illegal copies or modifications of the static program code image. Although most of these approaches include the instruction address none of them check the program flow. Therefore, ROP attacks at run-time are still possible and a secure program execution or a program code attestation are addressed by none of them.

6 Future Work

In a first prove of concept integration the tinyVLIW8 soft-core processor was extended by our approach. Since the processor is quite limited and not used in any common system an extension of the MSP430 is planned. Due to the von-Neumann architecture and the variable instruction format a more complex implementation of the MDU is necessary. Nevertheless, we are convinced that the logical overhead is quite moderate and the presented approach is still suitable.

Beside a hardware integration of our approach in an MSP430 soft-core an adaptation of the nonce will be investigated. The current approach causes a significant memory overhead on an MSP430. Each instruction expanded by a nonce gets an address of the limited address space. But a nonce is necessary for non-sequential instructions only. Therefore, we investigate the building of instruction blocks instead of single instruction as well as the usage of the instruction as nonce itself. But both approaches may have its own drawbacks and must be analyzed carefully.

In this paper we did not consider side-channel attacks, but they are highly interesting. Hence, we will investigate these effects on FPGA as well as on silicon devices with different MCU cores. Especially the current separation of the code execution and the block encryption may be an ideal entry point for side-channel attacks and must be analyzed.

7 Conclusion

In this paper we introduced the concept of intrinsic code attestation (ICA), shown its resistance against a wide variety of attacks and evaluated its overhead. ICA allows to execute encrypted instructions that are even depending on their predecessors. These features ensure that only authorized code can be executed. Decrypting non authorized instructions does not result in valid instructions. The chaining prevents reordering of instructions to implement an attack by "re-using" authorized instructions. These features allow a continuous protection of the devices, which sets our approach apart from earlier approaches that detect attacks only after the fact. Our simulations show that ICA comes at reasonable cost. The performance penalty can be as small as a single clock cycle. The related area overhead is then about 45 per cent for an MSP430 clone. The latter is somewhat significant if production cost is taken into account. But, if applications such automation control of energy distribution networks or similar sensitive applications are considered, the additional cost of the MCU are affordable and by far cheaper than costs resulting from a successful attack.

References

1. Running Leon2 on the Altera Nios Development Board, Cyclone Edition
2. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: ACM Conference on Computer and Communication Security (CCS), number MSR-TR-2005-18, Alexandria, VA, pp. 340–353, November 2005
3. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: A theory of secure control flow. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 111–124. Springer, Heidelberg (2005)
4. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: Proceedings of the IEEE Symposium on Security and Privacy, SP 1997, pp. 65. IEEE Computer Society, Washington, DC (1997)
5. Best, R.M.: Microprocessor for executing enciphered programs (1979)
6. Best, R.M.: Crypto microprocessor for executing enciphered programs (1981)
7. Best, R.M.: Crypto microprocessor that executes enciphered programs (2004)
8. Borghoff, J., et al.: PRINCE - a low-latency block cipher for pervasive computing applications - extended abstract. In: Wang, X., Sako, K. (eds.) Advances in Cryptology – ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg (2012)
9. Candelore, B., Sprunk, E.: Secure processor with external memory using block chaining and block re-ordering (2000)
10. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 400–409. ACM, New York (2009)
11. Chen, L., Landfermann, R., Löhr, H., Rohe, M., Sadeghi, A.-R., Stüble, C.: A protocol for property-based attestation. In: Proceedings of the First ACM Workshop on Scalable Trusted Computing, STC 2006. ACM, New York (2006)
12. Chen, X. Dick, R.P., Choudhary, A.: Operating system controlled processor-memory bus encryption. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2008, pp. 1154–1159. ACM, New York (2008)
13. Deng, J., Han, R., Mishra, S.: Secure code distribution in dynamically programmable wireless sensor networks. In: Proceedings of the 5th International Conference on Information Processing in Sensor Networks, IPSN 2006, pp. 292–300. ACM, New York (2006)
14. Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Anguille, C., Bardouillet, M., Buatois, C., Rigaud, J.B.: Hardware engines for bus encryption: a survey of existing techniques. IEEE (2005)
15. Eldefrawy, K., Francillon, A., Perito, D., Tsudik, G.: SMART: secure and minimal architecture for (establishing a dynamic) root of trust. In: Proceedings of 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, CA, USA, February 2012
16. Eriksson, J., Dunkels, A., Finne, N., Österlind, F., Voigt, T., Tsiftes, N.: Demo abstract: MSPsim - an extensible simulator for MSP430-equipped sensor boards. In: Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN 2008), Bologna, Italy, January 2008
17. Francillon, A., Castelluccia, C.: Code injection attacks on harvard-architecture devices. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 15–26. ACM, New York (2008)

18. Francillon, A., Perito, D., Castelluccia, C.: Defending embedded systems against control flow attacks. In: Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code, SecuCode 2009, pp. 19–26. ACM, New York (2009)
19. Gilmont, T., Legat, J.-D., Quisquater, J.-J.: Enhancing security in the memory management unit. In: Proceedings of the 25th EUROMICRO Conference, vol. 1, pp. 449–456. IEEE Computer Society (1999)
20. Girard, O.: openMSP:: Overview (2014)
21. Holcomb, D.E., Burleson, W.P., Fu, K.: Power-up sram state as an identifying fingerprint and source of true random numbers. IEEE Trans. Comput. **58**(9), 1198–1210 (2009)
22. Kil, C., Sezer, E., Azab, A., Ning, P., Zhang, X.: Remote attestation to dynamic system properties: towards providing complete system integrity evidence. In: IEEE/IFIP International Conference on Dependable Systems Networks, DSN 2009, pp. 115–124, June 2009
23. Krauß, C., Stumpf, F., Eckert, C.: Detecting node compromise in hybrid wireless sensor networks using attestation techniques. In: Stajano, F., Meadows, C., Capkun, S., Moore, T. (eds.) ESAS 2007. LNCS, vol. 4572, pp. 203–217. Springer, Heidelberg (2007)
24. Kuhn, M.G.: Instruction search attack on the bus-encryption security microcontroller ds5002fp. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **47**, 1153–1157 (1998)
25. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2004, p. 75. IEEE Computer Society, Washington, DC (2004)
26. Lie, D.: Architectural support for copy and tamper resistant software. Architectural Support for Programming Languages and Operating Systems, November 2000
27. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-flow checking by software signatures. IEEE Transactions on Reliability **51**(1), 111–122 (2002)
28. Panic, G., Basmer, T., Schrape, O., Peter, S., Vater, F., Tittelbach-Helmrich, K.: Sensor node processor for security applications. In: Proceedings of 18th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2011, Beirut, Lebanon, pp. 81–84, December 2011
29. Park, T., Shin, K.G.: Soft tamper-proofing via program integrity verification in wireless sensor networks. IEEE Trans. on Mobile Computing **4**(3), May 2005
30. Perito, D., Tsudik, G.: Secure code update for embedded devices via proofs of secure erasure. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 643–662. Springer, Heidelberg (2010)
31. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: Swift: software implemented fault tolerance. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2005, pp. 243–254. IEEE Computer Society, Washington, DC (2005)
32. Seshadri, A., Perrig, A., Doorn, L.V., Khosla, P.: SWATT: SoftWare-based ATTestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, USA (2004)
33. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, New York, NY, USA (2007)

34. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, pp. 298–307. ACM, New York (2004)
35. Spinellis, D.: Reflection as a mechanism for software integrity verification. ACM Trans. Inf. Syst. Secur. **3**(1), 51–62 (2000)
36. Stecklina, O., Methfessel, M.: A tiny scale VLIW processor for real-time constrained embedded control tasks. In: Proceedings of the 17th Euromicro Conference on Digital Systems Design, DSD 2014, Verona, Italy, August 2014
37. Suh, G.E., O'Donnell, C.W., Sachdev, I., Devadas, S.: Design and implementation of the aegis singlechip secure processor using physical random functions. Technical report, MIT CSAIL, November 2004
38. Texas Instruments, Dallas, TX, USA. MSP430 Programming via the bootstrap loader (BSL), slau319l edition, September 2014
39. Urban, R., Schölzel, M., Vierhaus, H.T.: Entwicklungsumgebung fr den compilerzentrierten Mikroprozessorentwurf (CoMet). In: Tagungsband Dresdner Arbeitstagung Schaltungs- und Systementwurf, DASS 2014. Fraunhofer Verlag (2014)
40. Usselmann, R.: AES (Rijndael) IP Core:: Overview (2013)
41. Vater, F., Langendörfer, P.: An Area Efficient Realisation of AES for Wireless Devices. it - Information Technology **49**, 188–193 (2007)
42. Venkatasubramanian, R., Hayes, J., Murray, B.: Low-cost on-line fault detection using control flow assertions. In: Proceedings of the 9th IEEE Conference on On-Line Testing Symposium, IOLTS 2003, pp. 137–143, July 2003