Protecting the copyright of an embedded device with cryptography

Thomas DETTBARN
Fraunhofer Institute for Integrated Circuits IIS, Erlangen, Germany

Abstract

Abstract– Embedded devices are small computers with a CPU, memory (for example SRAM), a mass-storage unit (an external Flash-ROM) and input-/output capabilities. With enough knowledge about the CPU, an attacker can deassemble the software, and learn all about the implemented algorithms, thus accessing and disclosing intellectual property of the developing company. He could modify the content, and resale the product under his own label. Or he could sell verbatim copies of the device, resulting in lost revenues and potential liability issues. Therefore some sort of protection method should be applied, e.g. encryption of the Flash content. This way, software updates can be given to customers on a regular basis, without giving too much secrecy away.

I. Introduction

Overview This paper is structured as follows: First, the principles of cryptography will be shown. Secondly, a way of enhancing an embedded device's boot-sequence by means of decryption will be discussed. Third, a cryptographic coprocessor will be designed. Fourth a real-life scenario with the DRM-IP prototyping board will be used as an example for an application of such a coprocessor. The paper closes by presenting ideas to increase the presented level of security even further.

Assumptions This papers makes the following assumptions:

- An embedded device's software is supposed to be undisclosed intellectual property
- The embedded device has SRAM which is inaccessible from the outside, e.g. via JTAG-ports
- The embedded device is of von-Neumann-architecture
- The contents of Flash-ROM can be extracted
- An attacker knows which components are used within the embedded device
- Alteration of the software by somebody else than the manufacturer is unwanted
- Though less severe, verbatim copying of the software is unwanted as well
- For convinience reasons, the customer should be allowed to apply new Flash-images as software updates to the embedded device

Terminology *Encrypting* a *plaintext* x, thereby transforming it into a *ciphertext* c is applying an encryption algorithm f to it:

$$f(x) = c \tag{1}$$

Applying the inverse f^{-1} to it is called *decryption*

$$f^{-1}(c) = x \tag{2}$$

Therefore, the function f has to be injective. Additionally, f should also be non-structure preserving and non-commutative. Usually, f is associated with a *key* ε for en- and f^{-1} with δ for decryption:

$$f(\varepsilon, x) = c \qquad f^{-1}(\delta, c) = x \tag{3}$$

so that for every $\varepsilon_1 \neq \varepsilon_2$ and $\delta_1 \neq \delta_2$

$$f(\varepsilon_1, x) \neq f(\varepsilon_2, x) \qquad f^{-1}(\delta_1, c) \neq f^{-1}(\delta_2, c) \tag{4}$$

If $\varepsilon = \delta$, the function f is called a *symetric cipher*, if $\varepsilon \neq \delta$, it is called *asymetric*. Asymetric ciphers like RSA[1] usually require more complex operations, and would be unsuitable for copyright purposes.
This paper concentrates on symetric ciphers. Additionaly, an algorithm associated with a key *key* is denoted as

$$f_{key}(x) = c \qquad f^{-1}_{key}(c) = x \tag{5}$$

instead of $f(key, x)$.
Two key-encryption Mathematically, the application of a second key to encrypt the ciphertext is a concatenation:

$$f_{key1}(x) = y \qquad f_{key2}(y) = f_{key2} \circ f_{key1}(x) = c \tag{6}$$

Ideally, f is not commutative, so the keys have to be reversed for decryption:

$$f^{-1}_{key1} \circ f^{-1}_{key2}(c) = f^{-1}_{key1} \circ f^{-1}_{key2} \circ f_{key2} \circ f_{key1}(x) = x \tag{7}$$

II. Embedded devices

Like all computers, embedded devices are roughly divided into two subgroups: The *Harvard-* and the *von-Neumann* architecture[2]. While Harvard-architectures use a strict separation of software and intermediate data, von-Neumann computers share their memory between those two. This allows greater design flexibilty, but it also requires an extra memory-control unit, which demultiplexes the address onto the physical compoments.

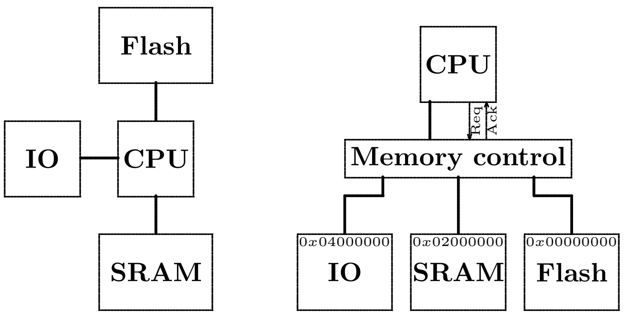

Fig.I: Left: A device, following the Harvard-architecture. Each component (Flash/SRAM/IO) is connected to the CPU via physically separate pathways. Right: A von-Neumann device. In this example, the Flash is mapped to address 0x00000000-0x01ffffff, SRAM to 0x02000000-0x03ffffff, and the Input-Output devices to 0x04000000 and up.

After sending a read- or write-request to the memory-controller, the CPU has to wait until this request has been acknowledged. Normally this is being realised as a simple `Req/Ack` handshaking protocol. The memory control unit takes care of all the components timing-constraints.

Booting Upon reset, the program counter within the CPU is typically set to an address within the Flash-ROM. Flash-ROM is relatively slow, so for speed reasons, its contents are mostly copied into the SRAM at boot-time[3].

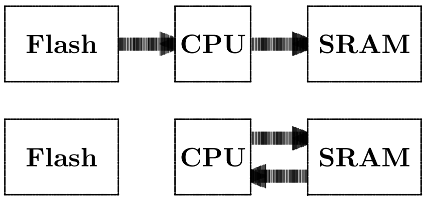

Fig.II: Top: during the boot-process, the program is loaded from the Flash-ROM and copied into the SRAM. Bottom: After the boot-process has been finished, the program is read and excecuted solemly from the SRAM.

Once the main program has been copied, an unconditional jump is being executed to the destination address in the SRAM. An example for such a bootloader, performing exactly that task is given in Src I.

```
void (*sram)(void)=0x02000000;
for (int i=0x00000000;i<0x00500000;i+=4)
{
    ((*volatile int)0x02000000+i)=
        ((*volatile int)0x01000000+i);
}
dataCacheFlush();
instCacheInvalidate();
sram();
```

Src I. A relativly simple boot loader: Parts of the ROM are copied into the SRAM. After that, the CPU is instructed to set the program counter to the SRAM. If the CPU is euqipped with cache, it is crucial that the SRAM is in a consistend state after booting. This is illustrated by the dataCacheFlush() and instCacheInvalidate() functions.

The for-loop performs the actual copying from a section within the Flash-ROM into the SRAM. Afterwards, the function `sram()` works as a placeholder for the first SRAM-address. To make sure that the program counter does not point to invalid memory after the jump, possible data and instuction caching needs considered. If a data cache exists, it needs to be sure that all data copied is actually written to SRAM. This is usually accomplished by "flushing" the data cache at the end of the copy loop. In addition, if an addition, if an instruction cache exists, this one needs to be invalidated, to make sure that correct and current program instructions are fetched.

Decryption while booting The fact that data and software share the same memory also has the secondary effect that a program can be modified before it is being executed. Possible scenarios for those modifications include the inflation of a compressed image, or the application of last-moment patches. Another possible modification can be used for copyright protection; by applying formula (2) to an encrypted image c within the Flash. The commands within the for-loop of Src I can be changed to the ones in Src II. Thus, c is being decrypted to the program x before being interpreted by the CPU.

```
((*volatile int)0x02000000+i)=
    finv(key,((*volatile int)0x01000000+i));
```

Src II. A modified bootloader, capable of decrypting.

Once the program x is in the SRAM, it has the same features as without the cryptographic booting sequence.

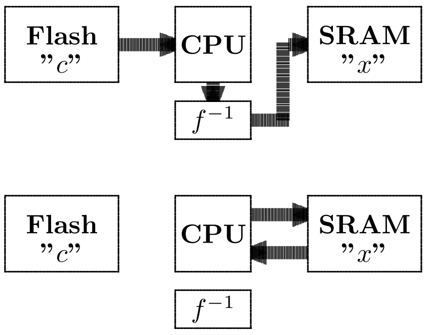

Fig.III: Top: Instead of copying the progam verbatim into the SRAM, it is rerouted through a decryption function. Bottom: Once the boot-sequence has ended its run, the program is run from the SRAM just like before.

Hence, the image within the Flash-image c is changed into an executable program x at boot-time. Without the proper key key, it is impossible to run or deassemble it. Moreover, it cannot be modified.

III. Serial numbers

In addition of protecting the software against unwanted analysation and modification, using different keys $key_1, key_2, \ldots, key_n$ in n embedded devices ties the Flash-image to a specific one. In doing so, it can not be transfered from one device to the other: According to formula 4, this would lead to an illegal program within the SRAM, which can not be executed.

ESN Certain types of Flash-ROMs have the feature of a uniqe electronic serial number (ESN), which is stored in a dedicated cell. Sending a special command-sequence to the Flash unlocks this cell for read-requests. For example, with Flash components manufactured by Spansion this sequence would be writing $0x00aa$ to address $0x0aaa$, then $0x0055$ to $0x0555$ and finally $0x0088$ to $0x0aaa$[4]. Every subsequent read-request is then rerouted to this cell. To exit, the sequence has to be repeated, but replacing the last write with $0x0090$ instead of $0x0088$. However, there are other types which have a write-once cell instead of a pre-written serial number. Flash-ROMs with this feature might be used to create a fake ESN. Those types can be identified by reading the ROMs features through the Common Flash Interface (CFI).

Higher protected keys The serial number alone would be a weak key. The Flash-ROM could be extracted from the embedded device and read relatively easy. Once in possession of the serial number $keyS$, an attacker could read the encrypted contents c in the Flash as well and write his own decryption algorithm

$$f^{-1}_{keyS}(c) = x. \quad (8)$$

This way he has unlimited access to the unencrypted program x. He could copy, analyse or modify it. Ergo, a second, higher protected key $keyH$ should be generated as well. This second key should be applied to x according to formula (6):

$$f_{keyH} \circ f_{keyS}(x) = c \quad (9)$$

Applying the two keys in the correct order is crucial: It can be easily observed that $f_{keyS} \circ f_{keyH}(x)$ results in weak copyright protection, as with this permutation, an attacker could perform the operation

$$f^{-1}_{keyS}(c) = f^{-1}_{keyS} \circ f_{keyS} \circ f_{keyH}(x) = f_{keyH}(x) = y \quad (10)$$

He could use the intermediate result y together with his own serial number $keyS'$ to create

$$f_{keyS'}(y) = f_{keyS'} \circ f_{keyH}(x) = c' \quad (11)$$

Even without the knowledge of $keyH$, he could clone the embedded device, replacing c with c'. At bootup, it performs an

$$f^{-1}_{keyH} \circ f^{-1}_{keyS'}(c') = f^{-1}_{keyH} \circ f^{-1}_{keyS'} \circ f_{keyS'} \circ f_{keyH}(x) = x$$

resulting in an useable device, running the program x, even though he is still unable to modify it. Therefore, the keys should be applied in $f_{keyH} \circ f_{keyS}(x)$ order. To replace the serial number, $keyH$ has also to be known to the attacker. However, a commutative f can cause the same problem.

IV. Designing a coprocessor

The decryption algorithm can be implemented in software. But, this would it make part of the (unencrypted) bootloader, thereby increasing its size. This approach also bears the danger of being deassembled: An attacker might get hold of the Flash-ROM-image c, deassemble the bootloader, and search for the hidden key $keyH$. With this knowledge, he could implement the decryption algorithm in formula (2) or (7) on his own, thereby giving him access to the encrypted part x. One possible solution would be to make c absolutely inaccesible to the customer. This also means that he will not be able to apply software updates to it in the future.

Decryption in hardware Another solution is to keep the key and the decryption algorithm completly out off the Flash-ROM, and thus off the CPU. This leaves a hardware implementation as the next logical option, which has also the advantage of being faster and it has a smaller memory-footprint. A cryptographic coprocessor, interconnected to the CPU via the memory-control unit is an efficent solution. Every operation inside this coprocessor will be shielded from the bootloader, and the CPU core itself does not have to be changed.

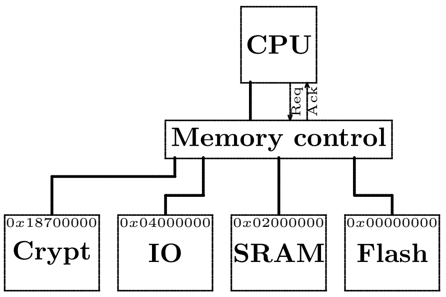

Fig.IV. A cryptographic coprocessor has been added to a von-Neumann computer. It has been mapped to the address $0x18700000$ and up through the memory control unit.

The CPU, instead of decrypting the blocks $c_1, c_2, \ldots, c_n$ on its own, hands them to the coprocessor. The unencrypted results $x_1, x_2, \ldots, x_n$ are then read back and can be processed otherwise.

Interfaces CPUs for embedded devices are either 8-, 16- or 32-bit wide. Modern cryptographic algorithmns require blocks of 128 bit or more to guarantee high-security protection. Meaning, that a register bank has to integrate a number of smaller values into a larger one before the coprocessor can conduct its operations.

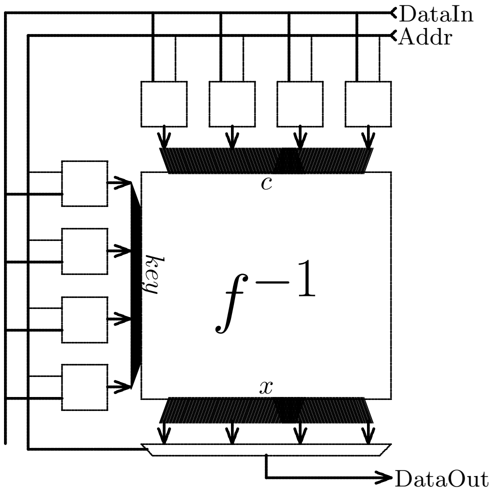

Fig.V. The internals of a coprocessor: The core f^{-1} requires 128-bit values for its operation. To connect it to a 32 bit CPU, the data has to be stored in 4 32-bit-registers first.

The simplest way to do so is by assigning an address range to the device through the memory control unit. Each address corresponds with a certain register. On software level, each of those can be written independently. Writing into the last register is the signal for the cryptographic core to start processing. Reading from the same address range returns the decrypted block. This also means that the core has to be able to hold its result until the last value has been read. On this side a multiplexer is used to determine which part of the result is returned upon a read-request. Fig V shows a schematic for such an interface. For a 32-bit-CPU the for-loop within the bootloader would look like the one Src III.

```
for (i=0x00000000;i<0x00500000;i+=16)
{
    ((*volatile int)0x18700000)=
        ((*volatile int)0x01000000+i);
    ((*volatile int)0x18700004)=
        ((*volatile int)0x01000000+i);
    ((*volatile int)0x18700008)=
        ((*volatile int)0x01000000+i);
    ((*volatile int)0x1870000c)=
        ((*volatile int)0x01000000+i);

    ((*volatile int)0x02000000+i)=
        ((*volatile int)0x18700000);
    ((*volatile int)0x02000004+i)=
        ((*volatile int)0x18700004);
    ((*volatile int)0x02000008+i)=
        ((*volatile int)0x18700008);
    ((*volatile int)0x0200000c+i)=
        ((*volatile int)0x1870000c);
}
```

Src III: A bootloader, reading the contents of the Flash-ROM (mapped at 0x01000000), sending it to the coprocessor (mapped at 0x18700000), and writing the results into the SRAM (mapped at address 0x02000000).

Four 4-byte words are read subsequently from the Flash (mapped to address 0x01000000) and send to the coprocessor (mapped to address 0x18700000). After writing the last register, the coprocessor starts working. Given that the memory-control unit is equipped with Request-/Acknowledge handshaking signals, the next instruction can read the result back from the coprocessor and store it inside the SRAM (mapped to 0x02000000), because the CPU is on hold until the coprocessor finishes.

Obtaining the key without the CPU By sending a special sequence of commands to the Flash-ROM, the CPU instructs it to return its serial number at the next read-request. If the coprocessor is designed to snoop on the connection between the CPU and the Flash, it could wait for this sequence, and read it off the bus as soon as it is returned into the CPU. Now it is impossible to send a fake ESN by changing the bootloader.

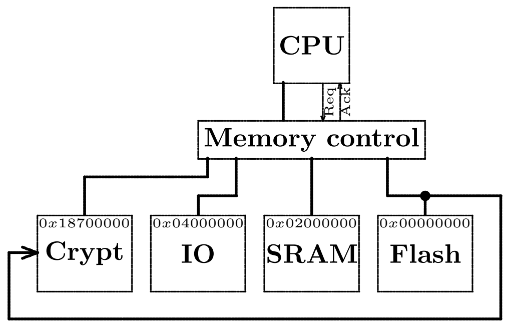

Fig.VI. By interconnecting the Flash-ROMs busses with the cryptographic coprocessor it is possible to read the serial number directly into the core.

A statemachine within the coprocessor waits for the access-sequence, which triggers a chip-enable for the key-registers. Fig. VII illustrates the reading of the first 32 bits of the key, in this case 0x12344f5e.

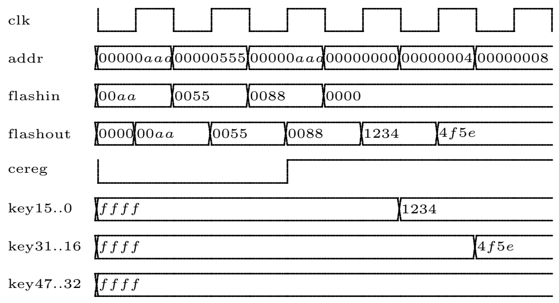

Fig.VII. After sending the sequence to switch the Flash to the ESN-area, a chip-enable signal cereg is triggered, and the serial number is stored within the coprocessors registers.

For obvious reasons, the instructions to switch the Flash to the ESN-area, reading the serial number, and returning to standard Flash-ROM have to be executed from a location within the SRAM. Executing this sequence from the Flash-ROM is impossible.

V. Rijndael-128 AES

Virtually any algorithm f^{-1} can be used in the decryption core, ranging from a simple XOR-array to a modern elliptic curve cipher. For a hardware implementation the Rijndael-128 algorithm as defined in [5] should be chosen, because it combines both a low complexity and a high protection level.

History In 1998 the Electronic Frontier Foundation presented a brute-force attack against the DES-algorithm[6, 7, 8]. The 56-bit cipher was broken in less than 60 hours. This was a dramatical demonstration that the federal encryption standard (which has been in place since 1976) became vulnerable. The shock of this presentation cumulated in a call for algorithms by the U.S. government for a new Advanced Encryption Standard (AES), which in turn would be a Federal Information Processing Standard (FIPS). It was a competition, consisting of two rounds. In the first round, which took place over a period of eight months, each algorithm was analysed in depth. The german contribution, MAGENTA[9], handed in by the Deutsche Telekom AG, did not advance beyond the first round and was in fact broken during the questionaire of its presentation[10].

The finalists for the second round consisted of MARS, RC6, Rijndael, Serpent and Twofish. In the end, Rijndael was adopted as the algorithm for the new standard.

The algorithm Rijndael is a block-cipher. Even though it is applicable to 192- and even 256-bit blocks, the standard only covers 128 bit blocks. However, the key sizes covered by AES are 128, 160, 192 and 256.

The encryption of a 128-bit block is performed in a number of rounds, depending on the keysize. In each round, the key is iterated and XORed with the block. Afterwards, the result is fed through 3 distinct functions (SubBytes, MixCols, ShiftRows), to further increase security.

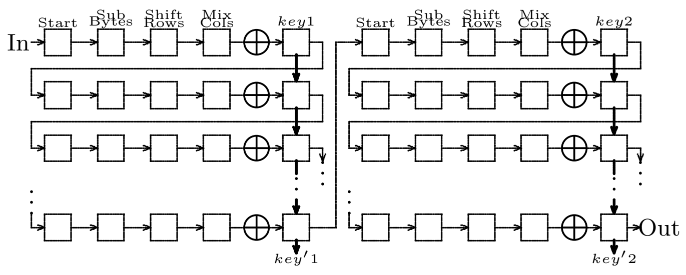

Fig.VIII. A graphical representation of Rijndael-128 AES with 2 keys

The standard, along with a graphical representation of the algorithm can be downloaded of the internet as FIPS-197. It should also be noted that Rijndael-256 has been adopted by the NSA as the encryption standard for TOP-SECRET data[11].

Key chainig After iterating the key eleven times for one block, the question arises what to do with it. Resetting the key after each decrypted block would lead to a vulnerability to statistical attacks, as blocks with the same content in x can be spotted in c as well. Iterating the key across block borders (called *key-chaning*) prevents this, but is only possible for sequential block-reads.

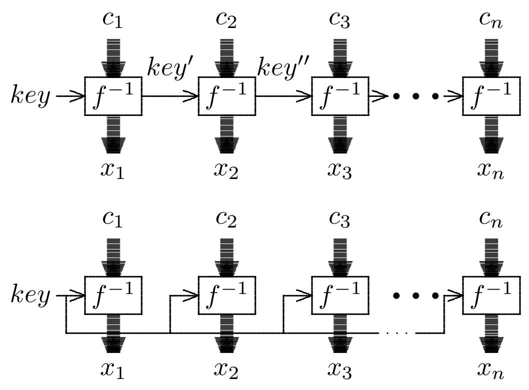

Fig.IX. Top: Chained key architecture, Bottom: Each new block resets the key

Instead of key-chaining, the address on where the block has been saved could be used as secondary key.

VI. Designing the decryption core

Rijndael was designed with the purpose of being easy to implement in hardware. Most of its operations can be performed by XOR-gates and bit-shifts. For a decryption core, the inverse functions have to be translated into Verilog or VHDL code. They are explained in great detail and with numerous examples in FIPS-197, and can be implemented as followed:

MixCols^{-1} An XOR-gate array performs this operation. For the first decryption round it has to be skipped, e.g. by a multiplexer.

ShiftRows^{-1} Being a Bit-permutation, this is a trivial operation in hardware and can be implemented by rerouting data-busses.

SubBytes^{-1} Even though the standard describes this operation as arithmetic, it should be implemented as a lookup-table or 16x256 bytes of ROM for speed reasons.

KeyIt The iteration of the keys can be implemented with 4x256 bytes ROM and an XOR-gate array.

According to the standard, KeyIt is not injective. For each block the keys have to be iterated in advance. In every round, those are 16 bytes which have to be stored. Rijndael-128 expects a minimum of 11 cipher rounds for safe encryption. Therefore, a total of 176 bytes have to be cached in an internal register-bank. A counter is used to control the contents: After it has been XORed, the key becomes obsolete and can be overwritten. Iterating the key and encrypting the text can be performed in parallel. The hidden key $keyH$ is hardwired into the core, and thus impossible to change or to be read externally.

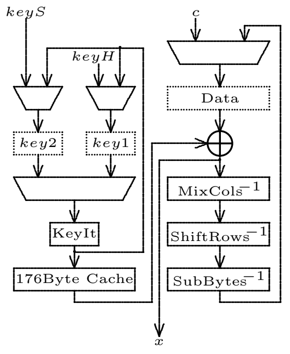

Fig.X. AES with 2 keys in hardware. The dotted blocks are registers, the other ones represent combinatorical blocks.

Fig. X shows a diagram for implementing the Rijndal-128-algorithm with two keys $keyS$ and $keyH$ in hardware. Depending on the programming of the two multiplexers in the top-left corner, the core can be used for chained- as well as for resetted key mode. Prior to the first round, the input c has to be stored into the `Data` register. For the very first block, $key1$ and $key2$ have to be initialised with the serial number $keyS$ and the higher protected key $keyH$. To counter side-channel attacks, the output x has to be buffered by an extra set of registers.

VI. Results

The Fraunhofer Institute for Integrated Circuits IIS used the presented design to enhance a prototyping board with copyright protection. The board[12] is capable of decoding signals encoded according to the Digital Radio Mondiale (DRM)-Standard[13]. For that, it uses two Xilinx Virtex-II and one Altera Excalibur FPGA. For demonstration reasons, the board has also been equipped with an ARM9-core, which handles the audio playback. Its software is confidential should not be disclosed. On the other hand, the nature of such a prototyping board requires constant updates. It would be impractical to exchange hardware each time a new feature has been added. With a cryptographic coprocessor aiding the boot-process, it was possible to such updates via the E-mail.

Time-complexity Our implementation of the decryption core is capable of performing one round per instruction cycle. It uses two keys. A whole 128-bit block therefore needs 22 cycles. Running at 25Mhz, this gives a theoretical throughput of 18.2MByte/s. The whole program for the ARM9-core fitted into 800kBytes, which could be decrypted within 21ms.

Gate-count The implementation on our prototyping board required the following number of NAND2-gates:

Table I: Gatecount

Block	XOR	SubBytes	MixCols	Cache	KeyIt
GateCount	223	9043	1485	5835	2534

With some additional optimizations in the synthesis-stage, the whole core cumulated in 16879 NAND2-gates.

Memory-consumption The coprocessor is completly encapsulated, its cache internal. To operate, it needs no space within the SRAM. However, due to the fact that Rijndael-128 works on 16-Byte blocks, the program in the Flash might have to be padded with up to 15 extra Bytes, to make its size divisible by 16.

Extended production time Owing to the fact that each device needs a unique Flash-image, an automated production becomes complicated. Once it has been assembled, the serial number has to be read. Afterwards the program has to be encrypted and stored within the Flash.

VII. Hardening security

The methods presented in this paper have shown a way of protecting the software of an embedded device against unwanted alteration and software piracy. To increase the level of security some extra measures can be taken into consideration as well.

Using two distinct decryption algorithms A different decryption algorithm can be used.

$$f_{keyH_1} \circ g_{keyH_2} \circ f_{keyS} \circ g_{keyS'}(x) = c \tag{12}$$
$$g^{-1}_{keyS'} \circ f^{-1}_{keyS} \circ g^{-1}_{keyH_2} \circ f^{-1}_{keyH_1}(c) = x \tag{13}$$

If one of the two algorithms should ever be broken, the program is still encrypted with the other one, leaving the protection intact.

Chaining the blocks By using the last decrypted block or a checksum of it as salt during decryption will chain the decrypted results to their predecessor. Consequently, changing blocks at random to figure out which of them are used to vital information will render all subsequent ones useless.

Encrypting the SRAM If the SRAM is not in the same package as the CPU, its contents should be encrypted as well. A second, transparent cryptocore between the memory-controller and the SRAM can be implemented. The read- and write-requests happen at random, and not sequentially. A chained key architecture can not be used. To counter statistical attacks the address can be used instead of *keyS* as the second key. Such a core must also be capable of encryption.

References

[1] T. Cormen, C. Leierson, R. Rivest, C. Stein, "Introduction to Algorithms (second edition)", pp881-887, ISBN 0262032937, MIT Press 2003

[2] W. Wolf, "Computers as Components: Principles of Embedded Computing System Design", ISBN 155860541X, Morgan Kaufman Publishers 2000, pp58-59

[3] A. Sloss, D. Symes, C. Wright, "ARM System Developer's Guide", ISBN 1558608745, Morgan Kaufman Publishers 2004, p13

[4] "Am29LV320D Datasheet", Spansion, July 11, 2005, pp30

[5] Federal Information Processing Standards, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)", fips-197, November 26, 2001

[6] Federal Information Processing Standards, "Data Encryption Standard (DES)", fips-46-3 (third revision), October 25, 1999

[7] W. Press, S. Teukolsky, W. Vetterling, B. Flannery, "Numerical Recipes in C++", pp304-308, ISBN 0521750334, Cambridge University Press 2003

[8] John Gilmore, "Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design", Electronic Frontier Foundation, ISBN 1565925203, O'Reilly Media 2000

[9] K. Huber, S. Wolter, "Telekom's MAGENTA algorithm for en-/decryption in the gigabit/sec range", ICASSP 1996 Conference Proceedings, volume 6, pages 3233-3235, 1996

[10] R. Weis, "AES und Attack", ISSN 0930-1054, CCC Datenschleuder 80, 2002

[11] The Comittee on National Security Systems, "National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information", CNSS Policy No. 15, Fact Sheet No. 1, June 2003

[12] F. Mayer, M. Schlicht, A. Heuberger, S. Melzer, "Chipset Development for Digital Radio Mondiale (DRM)", Proceedings of ICCE05, 10.-12.01.2005, Las Vegas, USA

[13] EBU, "Digital Radio Mondiale (DRM) System Sepcification (V 2.1.1)", ETSI Standard ES 201980, 2004

[14] B. Schneier, "Applied Cryptography", ISBN 0471117099, John Wiley and Sons 1996

[15] A. Menezes, P. Oorschot, S. Vanstone, "Applied Cryptography", ISBN 0849385237, CRC Press LLC 1997

[16] Anonymous, "All Hackers Need To Know About Elliptic Curve Cryptography", pp03, ISSN 1068-1035, Phrack Magazine 63, July 30, 2005