7.1-2

Using Cryptography as Copyright Protection for Embedded Devices

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

Abstract– **Copyright protection should not be limited to content alone. Software, running on an embedded device and stored in its flash-ROM, is also in danger of being copied or rebranded, resulting in lost revenues and liability issues. Cryptography is a way to prevent this, while being invisible to the customers.**

I. Introduction

Overview First, this paper shows a way of protecting an embedded device against reverse-engineering and unwanted alteration. The second part is about personalising each device with a uniqe serial number. Finally, the Digitial Radio Mondiale[1] prototyping board will be used as an example for a hardware implementation.

Why cryptography? Software in embedded devices is stored in non-volatile memory components. Those bear the danger of being extracted, read and analysed, thereby disclosing intellectual property. Making it physically inaccessible to the customer has the side-effect that it prevents the installation of updates.

If the software is encrypted, it is shielded against unwanted analysation and alteration. A customer can still exchange it with an updated version.

Terminology Mathematically speaking, the *encryption* of a *plaintext* x is applying a function f to it:

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

Applying the inverse function f^{-1} on the *ciphertext* c is called *decryption*. Usually, the function f is associated with a *key*, so that

$$f_{key1}(x) \neq f_{key2}(x) \qquad \forall\ key1 \neq key2. \tag{2}$$

In addition to being injective, f should also be non-structure-preserving and non-commutative. If f and f^{-1} use the same key, it is called a *symmetric cipher*, in contrast to *asymmetric* ones, where en- and decryption need two different keys.

II. Embedded devices

Booting of an embedded device Embedded devices (e.g. cellphones, PDAs, Internet-routers) are typically equipped with Flash-ROM, on-chip SRAM and a CPU. Complex ones run a modern operating system like Linux or Windows Mobile. More basic systems (like MP3-players) execute a single program over and over again. Running a program from Flash is slow. So its content is copied into the on-chip SRAM by the bootloader.[2]

Fig.I: Booting of an embedded device. The Flash-ROM's contents are copied into SRAM, from where they are being executed afterwards.

Decrypting of the Flash At this point, the software is handled as data: It can be modified. More precise: A decryption algorithm can be applied to it. Instead of a program x, a ciphertext c (encrypted with a key $keyH$) can be stored in the Flash. This requires the operation

$$f_{keyH}^{-1}(c) = x \tag{3}$$

to be called at boot-time. Without knowledge of the key $keyH$, the program x can not be modified, even if c and f^{-1} are disclosed.

III. Cryptographic Coprocessor

Performing the decryption in software on the CPU is equally dangerous as leaving the ROM in plaintext: Enabling an attacker to analyse the algorithm, thereby identifying the key. Now he can implement the decryption on his own, giving him the rest of the ROM-image as plaintext. It is better to keep the key completely off the CPU, and the decryption algorithm along with it. This leaves a hardware implementation as the next logical option: Once it has been taped out, an integrated circuit's inner workings are next to impossible to analyse. Hardware implementations also have the secondary effect of a significantly shorter execution time. One design possibility is the creation of an extension to the CPU. Upon execution of a special assembler operation, this extension takes the ciphertext and returns it back as plaintext to the CPU once it has been decrypted. No intermediate results are accessible by the software. If the CPU is hardwired, a cryptographic coprocessor can be interconnected using the memory-controller:

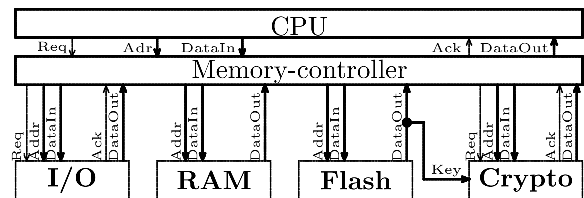

Fig.II: Attaching a crytographic coprocessor to a CPU through the Memory controller

According to the address, the memory controller redirects every read- and write-operation by the CPU to the coresponding device. Additionally, it implements a uniform handshaking protocol, so that on software level an Input-/Output terminal behaves exactly like a SRAM.

Accessing the coprocessor from software To access the coprocessor, a programmer has to know which addresses correspond with it. In Assembler or C a call would be:

```
sta 0x18700000 r1  //*((volatile int*)0x18700000)=r1;
lda r4 0x1870000c  //r4=*((volatile int*)0x1870000c);
```

Booting For instance, such a coprocessor could aid the booting process: Upon power-up, the CPUs internal program-counter is set to an address in the ROM. From there, it executes an unencrypted program that performs a for-loop a few bytes from the ROM, calls the coprocessor, and stores the result in the SRAM. This is continued until the whole ROMs contents are decrypted. Finally, the program-counter is set to the SRAM-address.

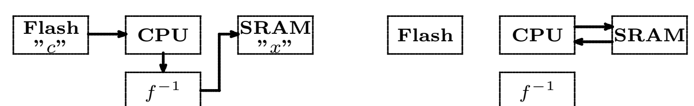

Fig.III: A coprocessor, conducting the operation $f^{-1}(c) = x$ is aiding the boot-process.

IV. Serial numbers

Using a serial number as secondary key changes the encryption chain to

$$f_{keyH} \circ f_{keyS}(x) = c \qquad f_{keyS}^{-1} \circ f_{keyH}^{-1}(c) = x. \tag{4}$$

Thus, alteration of the serial number would render the device useless.

Flash serial numbers Certain types of Flash-ROMs[3] can be purchased with a fixed serial number. It is written by the manufacturers and unique, and it cannot be cloned. Other types can be identified through the Common Flash Interface at run-time.

Key permutation Though it cannot be read, *keyH* can be cloned along with the device. Additionally, *keyS* and *c* can be extracted relatively easy from the hardware. *f* might be disclosed as well. This is why the permutation should never be $f_{keyS} \circ f_{keyH}(x)$. Otherwise, an attacker could generate his own serial number $keyS'$, and write a program, conducting the operations

$$f_{keyS}^{-1}(c) = f_{keyS}^{-1} \circ f_{keyH} \circ f_{keyS}(x) = f_{keyH}(x) = y \tag{5}$$

$$f_{keyS'}(y) = f_{keyS'} \circ f_{keyH}(x) = c' \tag{6}$$

on the original ciphertext and store $keyS'$ along with c' in his cloned device. At boot-up, it performs the calculation

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

Even without knowledge of the forementioned hardware key *keyH*, or the unencrypted program *x* in the SRAM, the device will be usable without an "official" key. Effords to prevent cloning are negated. The same problem arises when the decrypting function is commutative.

To further increase security, the serial number can be sent to the coprocessor by interconnecting it to the ROM. Thus, it is impossible to send a fake serial number to the coprocessor, e.g. by altering the (unencrypted) bootloader.

V. Example of a cryptographic coprocessor

For a prototyping board, we opted for a coprocessor based upon the *Rijndael-128* algorithm, described in the Advanced Encryption Standard (AES[4]). This algorithm has a time-efficency in $\Theta(n)$. It is deemed to be patent-free.

Rijndael-128 The algorithm is a block-cipher. Encryption takes 16 bytes of plaintext, and transforms it into 16 bytes of ciphertext, by going through 11 *rounds*. In each round, a key (16 bytes long), is iterated and XORed with the plaintext. The result is fed through 3 distinct bijective functions (called SubBytes, ShiftRows, MixCols).

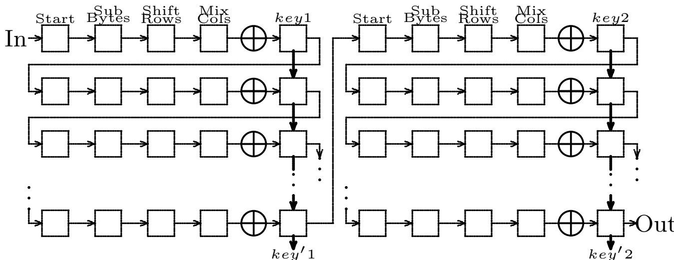

Fig.IV: AES-128 with two keys. Upon completion of all 22 rounds, $key1'$ and $key2'$ are used as new keys for the next 16 bytes.

To decrypt the ciphertext, the coresponding inverse functions have to be applied in reverse order.

Hardware-implementation Fig.V shows the schematic of the coprocessor. MixCols^{-1} is an XOR-array, KeyIt and SubBytes^{-1} are implemented as ROM lookup-tables.

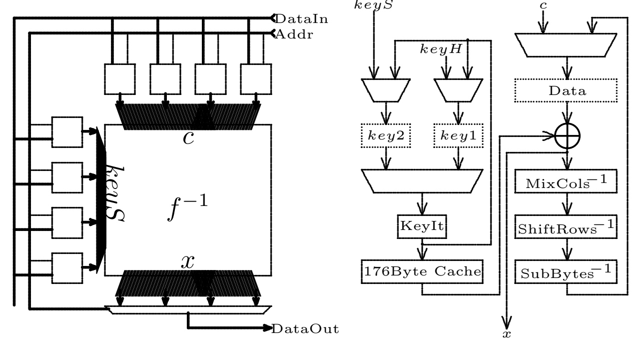

Fig.V: Left: The interface needed to make the Rijndael-128 core compatible to a 32-bit CPU. Right: Circuit of the core. The dotted blocks are registers, the other ones represent combinatorical blocks. The cache stores the iterated keys for 11 rounds. Note that *x* has to be buffered to counter side-channel attacks.

The results of the key iteration are stored in 176 Bytes cache, so that iteration and decryption can be performed in parallel. To counter side-channel attacks, the output is delayed through an extra register.

VI. Results

Latency Our coprocessor performs one AES-round per cycle. It uses two keys, so one 16 byte-block needs 22 clock cycles. At 25 Mhz, this gives a theoretical throughput of 18Mbytes/s.

Gatecount The implementation on our prototyping board required the number of gates given in table I.

Table I: Gatecount

Block	XOR	SubBytes^{-1}	MixCols^{-1}	Cache	KeyIt
GateCount	223	9043	1485	5835	2534

ShiftRows^{-1} is a trivial function in hardware, resulting in 0 gates. With some additional optimizations in the synthesis-stage, the core cumulated in 16879 NAND2-gates in 90nm technology.

Memory consumption Due to the fact that AES needs 16 byte blocks as input, the unencrypted image had to be rounded up to the next factor of 16.

Conclusion As this paper has shown, a cryptographic coprocessor can be implemented with a small gate count. It is reasonably fast, uses no extra memory, and prevents alteration and cloning. Moreover, it is completely transparent for the standard customer. It would only be recognized by an attacker. New updates can be sent through insecure open channels (like the internet) and applied by the customers themselves.

References

[1] EBU, "Digital Radio Mondiale (DRM), System Specification (V2.1.1)", ETSI Standard ES 201980, 2004

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

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

[4] Federal Information Processing Standards, "Announcing the Advanced Encryption Standard (AES)", FIPS 197, 2001

[5] B. Schneier, "Applied cryptography", ISBN 0471128457, John Wiley& Sons 1996

[6] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, "Handbook of applied cryptography", ISBN 0849385237, CRC Press LLV 1997