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 (DRM) 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 inaccessible to the customer is no solution, because this also 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 of 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 (for example cellphones, PDAs, Internet-routers) are typically equipped with a flash-ROM, SRAM and a CPU. Complex ones run an operating system like Linux or Windows Mobile. More basic systems (like MP3-players) execute a single program over and over again.

Fig.I: Booting of an embedded device

Because running a program from the ROM is slow and energy-consuming, its content is copied into the on-chip SRAM at boot-time. Once there, it can be handled as intermediate data, which can easily be modified. More precise: A decryption algorithm can be applied to the program, before it is executed.

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.

III. Cryptographic Coprocessor

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 side-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:

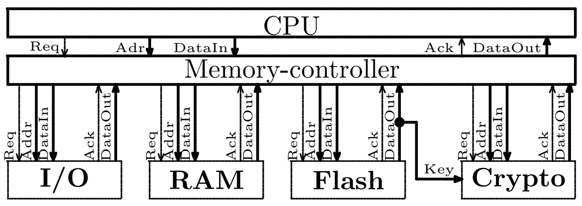

Fig.II: Attaching a crytographic coprocessor to a CPU through the Memory controller

According to the address, the memory controller redirects every read- and write-operation by the CPU to the coresponding device. On software level, an Input-/Output terminal behaves exactly like a SRAM, except for the number of clock cycles, given that they all use the exact same handshaking protocol.

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 look like this:

```
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.

IV. Serial numbers

Flash-ROMs can be purchased with a serial number. This serial number is written by the manufacturers and unique, and it cannot be cloned. This serial number can be used to personalize each flash-image through a second cryptographic encapsulation.

Two-key encryption Using the serial number as a second key changes the encryption chain to

$$f_{key1} \circ f_{key2}(x) = c \qquad f^{-1}_{key2} \circ f^{-1}_{key1}(c) = x. \tag{3}$$

Thus, alteration of the serial number would render the device useless. However, the serial number is publicly known. As might be the crypted program c. So the serial number should always be used as $key2$. Otherwise, given that the encryption method f is disclosed, an attacker could generate his own serial number $key3$, and write a program, conducting the operations

$$f^{-1}_{key1}(c) = f^{-1}_{key1} \circ f_{key1} \circ f_{key2}(x) = f_{key2}(x) = y \tag{4}$$

$$f_{key3}(y) = f_{key3} \circ f_{key2}(x) = c_2 \tag{5}$$

on the original ciphertext and store $key3$ along with c_2 in his cloned device. At boot-up, it performs the calculation

$$f^{-1}_{key2} \circ f^{-1}_{key3}(c_2) = f^{-1}_{key2} \circ f^{-1}_{key3} \circ f_{key3} \circ f_{key2}(x) = x. \tag{6}$$

Even without knowledge of the forementioned hardware key $key1$, or the unencrypted program x in the SRAM, the device will be usable without an "official" key. 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 the DRM prototyping board, we opted for a coprocessor based upon the *Rijndael AES-128* algorithm. This algorithm has a time-efficency in $\Theta(n)$. It is deemed to be patent-free.
AES-128 Rijndael 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), to further increase the level of security.

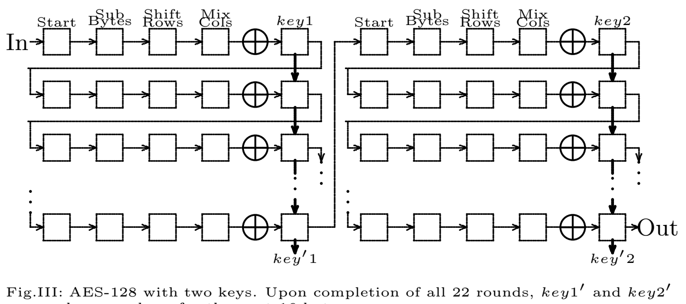

Fig.III: AES-128 with two keys. Upon completion of all 22 rounds, $key1'$ and $key2'$ are used as new keys for the next 16 bytes.

To decrypt the ciphertext, those functions have to be applied in reverse order.
Hardware-implementation Fig.? shows the schematic of the coprocessor.

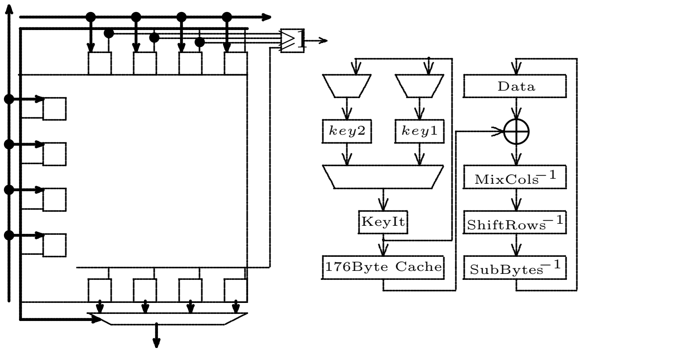

Fig.IV: Left: The interface needed to make the AES-128 core compatible to a 32-bit CPU. Right: Circuit of the core. The inverse functions are. The cache stores the iterated keys for 11 rounds.

The prototyping board is equipped with an Altera Excalibur, with an ARM-FPGA-combination in one package.
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 has to be 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. Note that the cache has to be filled for the first block. In this special case, decryption takes 33 cycles.
Gatecount The implementation on our prototyping board required the following number of gates:

Table I: Gatecount

Block	XOR	SubBytes	MixCols	Cache	KeyIt
GateCount	128	12345	12345	12345	12345

ShiftRows is a trivial function in hardware, thereby resulting in 0 gates. The whole core cumulated in 12345 gates, including 12345 gates for the handshake-interface. In 65nm technology, this is the equivalent of $0.04mm^2$ chiparea.
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 within a small chip-area. It is reasonably fast, uses no extra memory, and prevents alteration and cloning. On top of that, it is completely transparent for the standard customer. It would only be recognized by an attacker.

References

[1] J. Daemen and V. Rijmen, "The Design of Rijndael", ISBN 3540425802, Springer 2001
[2] A. S. Tannenbaum, "Modern operating systems", ISBN 0130313580, Prentice-Hall 2001
[3] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, "Handbook of applied cryptography", ISBN 0849385237, CRC Press LLV 1997
[4] Federal Information Processing Standards, "Announcing the Advanced Encryption Standard (AES)", FIPS 197, 2001