Project 2

Manzoor Ahmed

Pierre Vachon

First Draft

Power attacks have been a known threat since at least the publication of Kocher et al. (1999). An ordinary implementation of security algorithms could prove quite vulnerable – and in some cases, entirely transparent – when a sophisticated attacker has access directly to the circuits. The operations of some algorithms comprise conditional branching and other tell-tale signs that can be clearly visible from the point of view of power consumption. A well-designed security architecture should therefore take into account such considerations and seek to prevent possible attacks, under the assumption that an attacker will be able to measure power consumption. At the same time, the very nature of embedded devices means that the operation thereof, including security features, should not consume to much electricity, as batteries of limited capacity are often involved.

The article by McCann et al. (2015) does not address power attacks but is rather preoccupied with power consumption and limited memory space. The authors present an innovative process combining linear and non-linear polynomials that can generate a stream of bits for encryption at very low power costs, and requiring only a small amount of memory. Their solution demonstrates the feasibility of implementing complex security features in highly constrained environments.

Hell et al. (2004) address the issue of power attacks directly. Using statistical modeling of power consumption and a reduced instruction set, the authors develop an encryption protocol that operates at a level of power consumption that is constant enough to preclude power attacks. That is, by judicious use of select instructions with more or less known power consumption, they manage to naturally hide the different operations taking place during encryption. The authors do however note an important problem with their approach: the power consumption is much increased – at a minimum by 100%.

Our goal is to develop a workable security architecture that can resist power attacks at the same time as it consumes only a moderate amount of power. Given the findings of Kocher et al., we aim to create an encryption mechanism where neither simple power analysis nor differential power analysis is possible in reasonable time. At the same time, the power consumption of the algorithm should be a primordial concern so as not to hinder the very operation of embedded devices that may employ the algorithm.

To do this, we propose an implementation that blurs the power consumption profile of an algorithm by introducing some ‘noise’. By this, we mean that we will attempt to render any measurement of power fruitless in deciphering the actual operations taking place in the system. If Hell et al. have managed this already, we will try to achieve comparable results (security-wise) without the need for twice the power. We believe this can be achieved by inserting pseudo-random noise that would mask the operations but not consume as much power. The hoped-for result would exhibit a non-constant profile of power consumption that would somehow not betray the numerical values implicated in the calculations.

The challenge in accomplishing this is that inserting a merely random noise would render the system vulnerable to differential power analysis: running the same query on the system several times would allow an attacker to just take the average of the power consumption profiles to get the actual power profile. At the same time, inserting noise in a systematic, predictable fashion is also fruitless, as repeated attacks could very well reveal the noise inserting function. What we actually need is a function that, given a value, will yield a profile of noise that is seemingly random but somehow always the same for a given value. With this in mind, we plan on using a Pseudo Random Number Generator to produce a noise pattern. The advantage of this is that entering the same key value again and again would always generate the same power consumption profile, a result of the true power consumption and the noise. Each key would also generate a different power consumption profile as a function of the PRNG, which produce unpredictable values. The overall power consumption will therefore appear to have random spikes, uncorrelatable to the actual spikes of the basic power consumption profile. We intend to translate the results of the PRNG into noise by making use of superfluous instructions (junk code) based on a set of basic instructions (somewhat inspired from Hell et al. In sum, we aim to get an unbreakable power consumption profile that consumes less power than the constant model of Hell et al., thus accomplishing both goals: power security and low power consumption.

**Architecture and Design**

We are not able from a technical standpoint to measure power consumption directly. Our solution will therefore be preoccupied with CPU usage, as measured by available tools. With this in mind, we have elected to implement an ordinary version of RC4 in Python. We monitor some key metrics of the computer by using the functions CPUStat, MemoryStat, and HardDriveStat. These functions allow us to take repeated snapshots of these metrics during the operation of the RC4 implementation. Plotting the repeated snapshots with the help of the function matplotlib.pyplot shows some useful information on CPU usage.

The basic version of RC4 can serve as a comparative tool to evaluate the efficacy of our modified version. The modified version uses the Pseudo Random Number Generator described below to call junk instructions, for the purpose of masking via confusion the power consumption of the RC4, thus preventing Power Analysis. We have not yet determined the optimal fashion of integrating the noise generation into the basic implementation; we are currently trying to assess what density of instruction throwing would serve our goals best. We are exploring the possibility that insertion of a limited number of calls to the masking function at only a few strategic points in the basic implementation could be sufficient to thwart attacks.

The ultimate form of our modified RC4 would this incorporate these noise-generating function calls but would otherwise function as a normal implementation would, allowing for encryption or decryption. Some cost in time in power would be expended, but the result would be greater security.

Pseudo Random Number Generator

To implement our proposed solution, we make use of a pseudo-random function we have ourselves devised. This function seems to meet the requirements we have defined of providing a continuous stream of values but in a deterministic fashion, so that replaying the same key will result in the exact same noise profile being generated.

Our function takes in a key value – the value with which the attacked is trying to encrypt or decrypt. We first hash this key to create a new string seemingly unrelated to the original key. We have not yet settled on a given hash function, but any ordinary function meeting the basic requirements of hashing (collision resistance, most of all) would be suitable. This new string, along with a hardcoded table of junk instructions (used to generate the actual noise) are the two main ingredients of our function. The function operates on these two items by keeping track of two counters, and can run indefinitely.

The operating steps follow the repeating sequence. Extract the first seven binary digits of the new string, starting at the pointer value (initialized at 0). If too few digits are left, wrap around to the beginning using the modulo operator. Transform these seven digits to the corresponding number. Add this number to the current pointer (initialized at 0) of the instruction table and run the corresponding instruction. The instruction table pointer holds on to that value (also with a wrap around procedure for when the value exceeds the table size). The new string counter is updated by 7, and the bits corresponding to values 0, 2, and 5 of the 7-long substring we extracted are flipped. This is the terminal state of one iteration of the noise generating function, and the starting state of the next round, in which we will grab the next 7 bits of the new string, and apply that to the new table counter value.

The following shows the hypothetical process, with a key already hashed and counters starting at 0, with an Instruction Table size of 31. At each step, the modification is shown in bold.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Working Key | Work. Key Counter | 7 bits | Table Counter | Instruction |
|  | **110011001010** | 0 |  | 0 |  |
|  | 110011001010 | 0 | **1100110** | 0 |  |
|  | 110011001010 | 0 | 1100110 | **102mod31** |  |
|  | 110011001010 | 0 | 1100110 | 9 | **9** |
|  | 110011001010 | **7** | 1100110 | 9 |  |
| End of round 1 | **011001001010** | 7 | 1100110 | 9 |  |
|  | 011001001010 | 7 | **0101001** | 9 |  |
|  | 011001001010 | 7 | 0101001 | **(41+9)mod31** |  |
|  | 011001001010 | 7 | 0101001 | 19 | **19** |
|  | 011001001010 | **(7+7)mod12** | 0101001 | 19 |  |
| End of round 2 | **111001011110** | 2 | 0101001 | 19 |  |

For this function to work properly, it is bets to pick a new string size that is not a multiple of 7, and to pick an instruction table size with a primary number. We believe this architecture produces sufficiently random values as to preclude attacks.

**Current State of the Project and Upcoming Developments**

The number-generating part of the PRNG apparatus is functioning properly, and the first tabulations we have performed are encouraging. We are now looking to perform a systematic, structured test of the randomness that results, to ensure robustness against attacks.

The basic implementation of RC4 we have written is performing as expected. The CPU usage measurements resemble the power consumption graphs we have encountered in our literature review of the topic of Power Analysis; this is encouraging. The memory charts have however proved disappointing and may end up being of little tangible use.

The most pressing future developments are undoubtedly the selection of a junk instruction set and the integration of the noise function call to the RC4 routine. Our efforts have not yet revealed the right density of noise function calls to insert into the basic implementation. To select and settle on a proper level, we need to run more tests with attention focused on timing and CPU usage. Excessive time spent on the noise call function could prove too costly to make the system practical. But barring that, we are considering running some tests to ascertain optimal density.

Beyond this, we are considering the feasibility of try to perform an attack on the basic implementation of RC4 and on the modified version, to test whether or not an actual gain in security occurs. The format of such an attack is not yet clear to us yet.

**REFERENCES**

Hell M., Johansson T., Meier W. (2004). Grain - A Stream Cipher for Constrained Environments, *available at* <http://www.ecrypt.eu.org/stream/p3ciphers/grain/Grain_p3.pdf>

Kocher P., Jaffe J., Jun B. (1999). Differential Power Analysis, CRYPTO '99 Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology

McCann D., Eder K., Oswald E. (2015). Characterising and Comparing the Energy Consumption of Side Channel Attack Countermeasures and Lightweight Cryptography on Embedded Devices, IEEE, Secure Internet of Things (SIoT), 2015 International Workshop on