Power-Based Side-Channel Attack for AES Key Extraction on a ATMega328 controller

Utsav Banerjee, Lisa Ho, and Skanda Koppula

***Abstract*—We demonstrate extraction of a private-key from Flash program memory on the ATMega328 microcontroller (the controller used on the popular Arduino Genuino/Uno board). We loaded a standard AVR-architecture AES implementation onto the chip and ran this implementation to encrypt 500 randomly chosen plaintexts. By carefully measuring the chips power consumption, we were able to correlate the consumed power with the input plaintexts and key values that might be used to encrypt each AES block, and back-derive the hard-coded key used for encryption. We describe here our test infrastructure for automated power trace collection, an overview of our correlation attack, sanitization of the traces and interesting stumbling blocks encountered during data collection and analysis, and the results of our attack.**

***Index Terms*—AES, side-channel, power consumption, ATMega328, Correlation Power Analysis**

I. Introduction

Recent concerns about data privacy has brought attention to the necessity of encryption algorithms. One of the more popular symmetric-key algorithms, Advanced Encryption Standard (AES), has been the U.S. government standard since 2002 (ISO/IEC 18033-3), and is used in a multitude of applications: SSL/TLS protocols [1], Kerberos [2], and demonstrably secure embedded devices [3]. This last application in particular, embedded devices, has seen much growth in recent years, given the advantages of computation on smaller embedded devices: low power, lower system latency, and generally smaller device size.

Small hardware implementations, however, are notoriously vulnerable to a range of side-channel attacks [4]. Timing, electromagnetic radiation, and power consumption are just three commonly exploited vectors used to leak information about ongoing computations and data on the chip. Knowing that a device architecture is amiable to side-channel exploitation is useful in deciding whether to execute unprotected sensitive computations or store data on devices with similar memory and processor characteristics.

We aim to demonstrate a reasonably realistic side-channel attack on AES on one such embedded device: the ATMega328 microcontroller produced by Atmel. The ATMega328 is the basis for the widely popular development board, Arduino Uno [1]

In section two, we review for the read the theoretical ideas underpinning our attack. In section three, we describe our implementations our hardware setup, power measurement infrastructure, correlation methodologies, instructive problems that we encountered, and overview the structure of our source code. In section four, we describe quantitatively describe the results of our attack.

II. Preliminaries

A. Controller Specifications

The ATMega328 family of chips is an 8-bit microcontroller series with 32 KB of NAND-type flash and 2KB of SRAM. The controller runs off a 16 MHz external clock on the Arduino board. Typical power consumption of the chip ranges from 7 to 12V, with a 20mA current draw, depending on the peripheral and I/O pin usage [7]. Out attack exploits the NAND-type flash memory architecture that consumes marginally more power when accessing addresses that store value-zero (discharge) bits [2] [8].

The encryption program running on our ATMega328, `AESLib`, uses an Arduino-specific port of the `avr-crypto-lib` by Davy Landman and Bochum Hackerspace [5] [6]. `AESLib` is one of the more widely-used AES implementations for Arduino, and includes support for ECB and CBC-modes of AES. Our team decided that ECB-mode would be more amiable to a power correlation attack, and correspondingly chose to exploit the library's AES-ECB implementation. We discuss ECB in further depth in section 3.D.

B. Correlation Power Analysis

Correlation Power Analysis (CPA) is a type of side channel attack that relies on power consumption information. On a high level, CPA attempts to correlate observed power consumption with expected power consumption. To a greater extent than more basic forms of power analysis such as Simple Power Analysis, CPA attacks are able to extract secret keys from noisy data. This requires collecting the power consumption

All authors are with the Department of Electrical and Computer Engineering, Massachusetts Institute of Technology, Cambridge, MA, 02139 USA

To contact the authors: `utsav@mit.edu`, `lisaho@mit.edu`, and `skandak@mit.edu`

Manuscript completed for 6.858 Computer Systems Security; completed on December 5, 2015

[1]Other models of the Arduino, such as the Arduino Mega and Arduino Genuino Micro use ATMega chips as well, that have a similar architecture to the ATMega328. It is possible that this attack could be adapted to those chips as well.

[2]In a highly simplified power consumption model, NAND-flash charges a central power line connected to a series of memory cells. Depending on the value stored in the accessed memory cell, the line is discharged or not. Thus, line requires data-dependent recharging.

of a device performing encryption over many different plaintexts with the same key. We can then build a power model that contains the expected power consumption of the device performing a particular operation during encryption over the given plaintexts with every possible key.

In our case, we used CPA against AES encryption. There are several operations during AES that can be good candidates to model power consumption for depending on the hardware implementation. Attacks tend to model power from the first round of AES because it is most clear in this round which bytes of plaintext and key are getting combined. Power analysis tends to target the XOR operation in the `AddRoundKey` or the SBOX substitution during the `SubBytes` operation, since these can consume predictable amounts of power. Different implementations use different estimates to correlate power consumption with, most commonly the Hamming Weight of the operation result or the Hamming Distance of the input and output of the operation. These may be correlated with the amount of power consumed because reading a ”0” or ”1” from memory may require a different amount of power.

Using CPA against AES requires collecting the amount of power consumed over hundreds of plaintexts. We then calculate the expected power consumption for each of these plaintexts with every possible key. Trying every possible key with AES is possible because a different byte of the key is used for every byte of the plaintext. This reduces our search space to 2^8 possibilities for each key byte, which allows us to build a power model that takes every one of these 2^8 key byte possibilities into account for each byte.

After building a power hypothesis consisting of the amount of power consumed for each possible plaintext with every possible key byte, we can calculate the correlation coefficients between these expected amounts of power and the amount of power observed during encryption. One advantage of CPA is that we don’t need to know exactly when during encryption our targeted operation occurs; we can calculate the correlation coefficient between the power hypothesis and the power trace (collected power consumption over time) at each point of the trace. We then take the key byte that gives us the maximum correlation coefficient as our best guess key byte. Repeating this process 16 times gives us best guesses for all 16 bytes of the key.

III. Protocols and Procedure

A. Data Collection Infrastructure

A central piece of our work was developing the serial-connection based data collection framework to feed plaintexts for encryption to the Arduino, and read the resulting power trace from the oscilloscope. An outline is given in Figure 1.

We needed a subset of the power trace that correlated strongly with the input plaintext and key. Specifically, we were interested in finding the portion of the trace that corresponded to the `xor(plaintext,key)` operation, and the the SBOX permutation taken over the `xor` result. In order to automate slicing to this part of the trace, we modified the `AESLib` SBOX assembly code to insert a flag a memory-mapped register that pulled up pin 13. Our oscilloscope triggered on

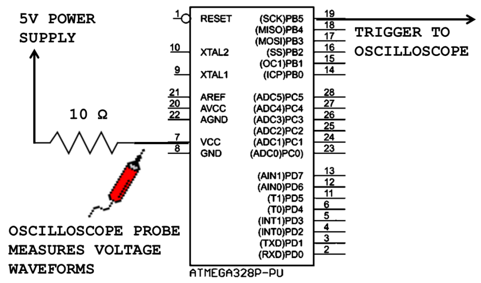

Fig. 1: The oscilloscope probes voltages on the chip’s power line, starting automated power trace on a trigger that signals the start of AES

this output, and automatically captured the part of the trace immediately after the pin 13 rising edge.

Central to our collection was a Tektronix 5054B computing oscilloscope that collected samples at 4GHz with internal memory bank of at most 16 million points. The sampling rate limited the resolution of traces that we were able to capture; we originally thought this physical cap on data quality was causing correlation issues we ran into, until we discovered arbitrary DC shifts in the traces. We discuss this challenge in more detail in Section 3.C. We used the oscilloscope’s GPIB query interface to automate configuration and downloading of traces.

In the final iteration of our system, we have a orchestrating computer C send plaintexts to the Arduino for encryption every 2 seconds over the Arduino’s serial port. The pin 13 trigger resulting in a oscilloscope trace capture, which was sent back to C.

More recently, we have been attempting to use the Tektronix $FastFrames$ feature to take capture batches of 2,500-point traces at once, and use one memory-read, GPIB-write operation to transfer these traces to the computer. This would allow us much faster trace measurement, which is currently bottlenecked by the GPIB-write operation. As we will discuss in Section 4, the number of key bytes we can recover is directly correlated with the number of plaintexts we can capture and average.

Photos of our collection framework can be found at `https://www.dropbox.com/sh/usialgelvfqlsdr/AACvqOHKEWoumWNYi2WRIebCa?dl=0`. Of perhaps small interest is the grounded aluminum box encapsulating the ATMega328 setup, which reduced the EM noise and interference which we noticed in our power traces.

B. Implementation of CPA and Power Model

We experimented with modelling the expected power consumption from both the XOR operation in the `AddRoundKey` step and the SBOX substitution from the `SubBytes` step of AES. To figure out which one would be more effective, we examined the power consumption of just the AVR assembly operations corresponding to these steps on the Arduino. We were unable to easily make out any difference in power consumption between when the result of the SBOX operation

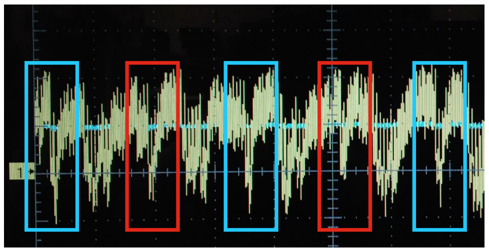

Fig. 2: XOR operations alternated between results of all "0"s and all "1"s. The power consumption was noticeably different depending on the Hamming Weight of the result.

was all "0"s or all "1"s, but we were able to easily see a diffference in power consumption between when the result of the XOR operation was all "0"s or all "1" (see figure below).

discuss different power models we've tried, faster correlation

In later iterations of our code, we aimed to speed up analysis of waveforms, and obtain results like that in Figure 3 more quickly; so we replaced our correlation function with a faster iterative correlation function that calculates the correlations given $1, 2, 3, \ldots, i, \ldots, num\_plaintexts$ traces, without recalculating correlations for the previous $i-1$ traces.

C. CPA Challenges and Solutions

[describe problem and solution for each one of these]

- Why we removed difAmp -¿ increase resistor value -¿ increase bandwidth?//used to measure current before
- averaging to solve dc shift
- remove serial print and interrupt to solve clock jitter and time shift/skew
- adding nops to prevent asyncronous / delay
- original hypothesis about sbox: SBox bad for correlation? the flash architecture -¿ bit block bit block bar
- Adding more data solved problems (link to results for more detail)

D. Drawbacks

Aspects of our attack implementation are unlikely to be possible in real-world attack scenarios; we address these concerns and suggest possible remedies.

– We chose to attack the Arduino library's implementation AES-ECB mode. Althought ECB is not semantically secure (i.e. you can derive information about the plaintext from the ciphertext), ECB is still (unfortunately) used as the default option in a number of crypto-suites, `avr-crypto-lib` included. This is because of it's relatively simple implementation, compared to other more sophisticated modes of AES. Furthermore, our attack does not exploit the plaintext-ciphertext correlations in ECB to derive the key; rather, it uses our power sidechannel. For our team's first power-trace based attack, we chose a mode that we were confident that might have some correlation with the plaintext-key-guess XOR (the first computation in the first step of ECB). It might be possible to adapt our attack to CBC-mode as well.

– We modified the `AESLib` assembly code to pull high one of Arduino's I/O to signal signal the oscilloscope where to begin collecting traces (the scope's trigger). In a real world attack, it unlikely that we will be able to insert a convenient signal to aid our trace collection and analysis result. Nonetheless, careful manual analysis of one trace could yield a constant time frame of interest, which, barring waveform time shifts, we could use to automate trace collection and trace slicing. We have demonstrated way to reduce time-shift errors in our work.

– We chose to collect traces from plaintexts that were purely alphanumeric (albeit chosen randomly within that). This solved issues we faced regarding transmission and rendering of special characters over the Arduino-computer serial port, which we needed to use as we were verifying that the Arduino was correctly executing AES. It is unlikely that this choice of plaintext adversely affected or falsely bossted our results. Nonetheless, future studies might be interested in investigating how choice of plaintext affects, if at all, our correlation power.

E. Overview of Source Code

All our source code can be found at `https://github.com/skoppula/aes-sidechannel`. Files in the source that might be of interest:

– `cpa/cpa.py` contains the source that reads in the waveform binary data (with helper `wfm2read_fast.py`) and subsequently run CPA. It outputs the top five key byte guesses per byte of key, and a correlation visualizations like Figure 3.

– `data-capture/arduino-aes-2` runs AES encryption on a new plaintext 10 times, everytime a 'run' command is received on the terminal. The ten traces are later averaged on the oscilloscope. `data-capture/arduino-aes-1` runs AES encryption on a plaintext sent over the serial port.

– `data-capture/scope-interface-2` runs a Python script that interfaces with the Arduino to run AES at regular intervals, and interfaces with the oscilloscope to collect the corresponding traces. `data-capture/scope-interface-1` is a Matlab implementation of similar functionality that we originally used to interface with the oscilloscope over GPIB; interfacing with the Arduino was done seperately in this version of the code (in `data-capture/send-plaintexts-processing`)

– `data-capture/process-wfms` contains the scripts we put together initially to parse and plot the first `*.wfm` waveform binaries that we recieved from the oscilloscope. This was later integrated into `cpa/cpa.py`

IV. Results

We found more robust results by triggering our traces to start from the beginning of the AES EBC-mode `xor(plaintext, key` and continue until $SBOX$ permutations are complete. These computations empirically correlated well with our key guesses for us. Furthermore, we found much higher rates of correct key guesses by disabling the chip's serial output and disabling interrupts. This seemed to eliminate inter-trace time-shifts that messed with our correlation analysis. In addition, averaging ten traces at a time seemed to eliminate DC power bias that shifted the power magnitude up and down between traces run on the same plaintext. Only after these corrections were we able to find any meaningful correlations and extract our first correct key guess.

We ran our system on three seperate times, with three different private-keys: two randomly generated keys and one non-random key [3]. In all three, we were able to discover the correct key.

However, interestingly, the number of traces required to get complete accuracy differed between the keys. Specifically, it took 600 trace averages (each corresponding to a plaintext) to get complete accuracy on the non-random key, 300 trace averages to completely uncover the first random key, and 400 trace average to uncover the second random. We were interested in understanding the performance of each key guess as the number of input plaintext traces increased, so for each run, we created a plot that tracked each key guess's correlation with the trace data as the amount of trace data increased (Fig. 3)

It takes roughly 30 minutes for our infrastructure to collect the traces for the encryption of 100 plaintexts. Data collection time seems to scale linearly with the number of plaintexts we process: iencrypting and collecting the traces for 500 traces, takes on average two and half hours. Processing these 500 traces, and extracting the correlation values and key estimates over i plaintexts for all i (Fig. 3) takes roughly five minutes.

Our data and results can be found at `https://www.dropbox.com/sh/07xni6s4tu4klme/AABnrBK-QZCVO1tMK4GFeQ5ta?dl=0`.

Acknowledgment

The authors would like to extend our deepest thanks to Chiraag Juvekar of the Energy Efficient Digital Circuits and Systems group for the time he spent with us aiding our debugging of data collection and analysis problems. We would also like to thank Albert Kwon, our TA, for his insightful advice over the course of the project.

References

[1] Lee, Homin K and Malkin, Tal and Nahum, Erich, *Cryptographic strength of ssl/tls servers: current and recent practices*. Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, 2007.
[2] Rathore, Romendrapal Singh and Pal, BL and Kumar, Shiv. *Analysis and Improvement in Kerberos 5*. 2015.
[3] Altera Corporation, *FPGAs with built-in AES: The key to secure system designs*. Embedded Computing Design, July 15, 2008.
[4] Oswald, Elisabeth, et al. *Side-Channel Analysis Resistant Description of the AES S-Box*. 2005.
[5] https://github.com/DavyLandman/AESLib. *Arduino AESLib*. 2015.
[6] http://www.das-labor.org/wiki/AVR-Crypto-Lib/en. *AVR-Crypto-Lib*. 2013.
[7] http://goo.gl/hhZF2z. *Atmel 8-bit Microcontroller Datasheet*. 2014.
[8] Handy, Jim. http://goo.gl/5QZHhK *3D NAND: Making a Vertical String*. 2013.

[3]By non-random, we mean the key `[0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF]`. This was the test key chosen during the development stage of our project, as we were building the trace-collection infrastructure and writing the CPA code.

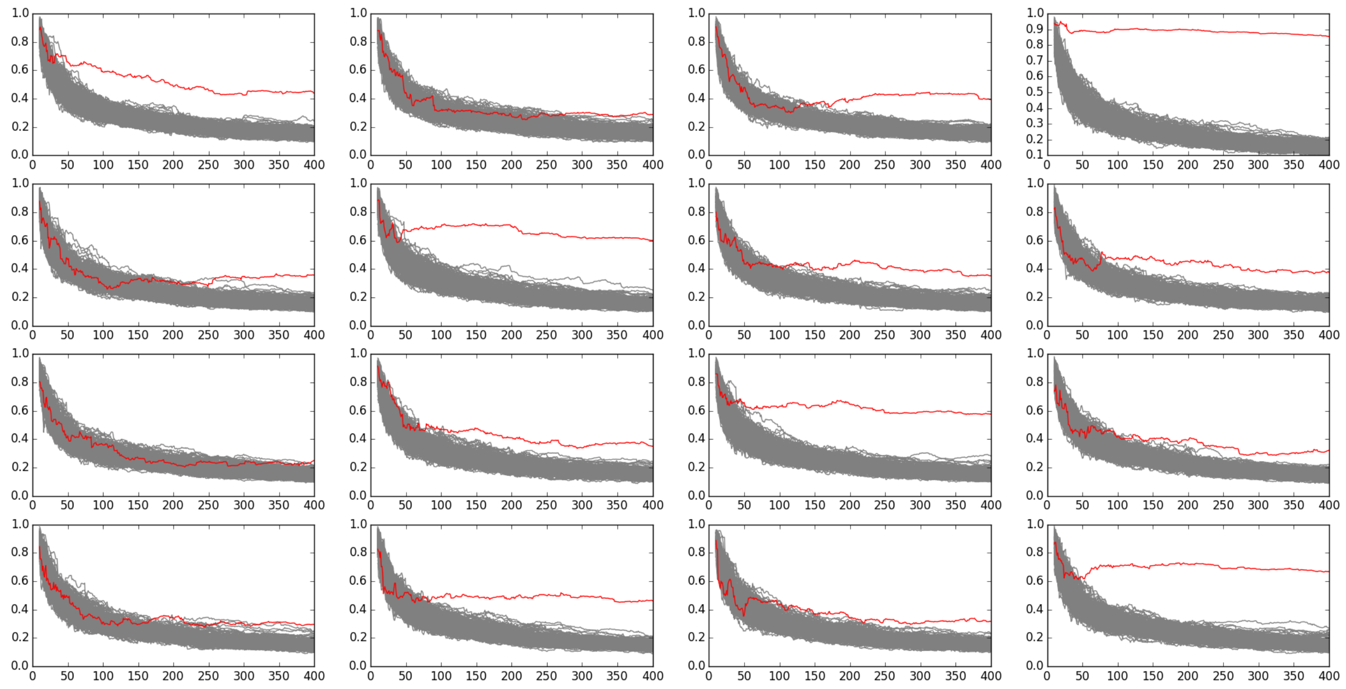

Fig. 3: Each graph corresponds to one of the 16 blocks/16 plaintext bytes/16 key bytes in our AES implementation. A single graph plots the correlation of all 256 key guesses (y-axis) with the power trace data, as more data is fed into the correlation (x-axis). The red line represents the correct key guess. Notice that by the time we feed the 400th trace into our CPA implementation, the correct key byte is the key guess with the highest correlation with all the traces (visible in the height of each red line at $x = 400$). Results for the other two CPA runs be found Dropbox linked below.