I, For One, Welcome Our New Power Analysis Overlords

An Introduction to ChipWhisperer-Lint

Colin O'Flynn & Greg d'Eon.
coflynn@newae.com
NewAE Technology Inc.
Presented at Black Hat USA, 2018.

August 7, 2018

You, standing there with the hardware accelerated microcontroller. Me, sitting with a slightly outdated Python install, equipment for performing power measurements on the microcontroller, and several browser tabs with parallel processing tutorials open. No time for getting into the backstory, how do we know if this will work before we've wasted a year of our life together? ChipWhisperer-Lint is currently a proof of concept to demonstrate that automated power analysis might not be such a crazy idea.

1 Introduction

This paper introduces the ChipWhisperer[1]-Lint tool, which is a proof of concept for automating side-channel power analysis leakage model detection. it also demonstrates how automated leakage detection could be used as part of a test suite for software or hardware development.

Note despite the LaTeX feel of this paper, it is not written as a proper academic paper. In particular, note that several of the tests have been done on a "one-off" basis. This means the attacks have NOT been averaged over multiple keys for example. The reader is carefully cautioned from attempting to directly use the reported numbers regarding "traces to break" and similar for the demonstration devices.

Instead, all source code of firmware used in recording the data, the raw data itself, and even the capture hardware and software is open-source. ChipWhisperer-Lint is posted at

[1] ChipWhisperer is a trademark of NewAE Technology Inc., registered in the United States of America, European Union, and other jurisdictions. All other product names, logos, and brands are property of their respective owners. All company, product and service names used in this document are for identification purposes only. Use of these names, logos, and brands does not imply endorsement.

`https://www.github.com/newaetech/ChipWhisperer-Lint`, and specific notes related to this article are posted at `https://www.github.com/newaetech/overlord-talk`.

1.1 What is Power Analysis

The following introduction is mostly reproduced from [9], and goes into (probably too much) detail about how power analysis works.

It had been known that the power consumed by a digital device varies depending on the operations performed since at least 1998, when Kocher, Jaffe, and Jun showed the use of the power analysis for breaking cryptography [6] and making money[2]. The first example given was that of Simple Power Analysis (SPA), where knowing the sequence of operations would directly allow read-out of the secret key. Differences in power consumption for different operations allows breaking of cryptographic algorithms using SPA.

As an example, consider the source code from Listing 1. This code is taken from the file `bigint.c` of `avr-crypto-lib`, an open-source library for the AVR microcontroller. This particular function is used as part of the RSA crypto system.

When a bit of the `exp` variable is 1 a square and multiply is performed, and when a bit of the variable is 0 only a square is performed. Looking at the power consumption, we can see some difference between a square and multiply operations. This is shown in Figure 1, where the code has been compiled onto an Atmel XMEGA microcontroller. The leakage in Figure 1 can be seen in the timing when a '1' is processed compared to a '0'. While both the square and multiply have similar power signatures on this platform, the delay on entering the square routine is slightly longer. The delay marked at "A" in this figure is about 80 mS, and the delay at "B" is about 60 mS. The slightly longer delay can very reliably be detected to determine if two function calls have occurred (square + multiply, indicating a '1') or only one function call (square, indicating a '0').

This particular variable that is leaked in this manner is not an arbitrary one, but instead knowledge of this variable leaks the value of the secret key used in this operation. Thus SPA allows us to directly break the secret key used during the operation.

While SPA is capable of breaking cryptography by deciphering operations, the same paper also presented a more powerful attack called differential power analysis (DPA) [6]. This seminal work demonstrated that there may be considerable problems with implementations of otherwise secure protocols on embedded hardware devices. In particular, this introduced the idea that measurements of the power could actually reveal something about the *data* on an internal bus, and not simply the overall operation.

Fundamentally, this is due to physical effects of how digital devices are built. A data bus on a digital device is driven high or low to transmit signals between nodes. The bus line can be modeled as a capacitor, and we can see that changing the voltage (state) of a digital bus line takes some physical amount of energy, as it effectively involves changing the charge on a capacitor.

[2]This part is more difficult, and considerably less repeatable.

```
uint8_t flag = 0;
t=exp->wordv[exp->length_W - 1];
for(i = exp->length_W; i > 0; --i){
  t = exp->wordv[i - 1];
  for(j = BIGINT_WORD_SIZE; j > 0; --j){
    if(!flag){
      if(t & (1 << (BIGINT_WORD_SIZE - 1))){
        flag = 1;
          }
    }
    if(flag){
      bigint_square(&res, &res);
      bigint_reduce(&res, r);
      if(t & (1 << (BIGINT_WORD_SIZE - 1))){
        bigint_mul_u(&res, &res, &base);
        bigint_reduce(&res, r);
      }
    }
    t <<= 1;
  }
}
```

Listing 1: The following lines are from `bigint.c` in `avr-crypto-lib`, showing an example implementation of the vulnerable RSA code.

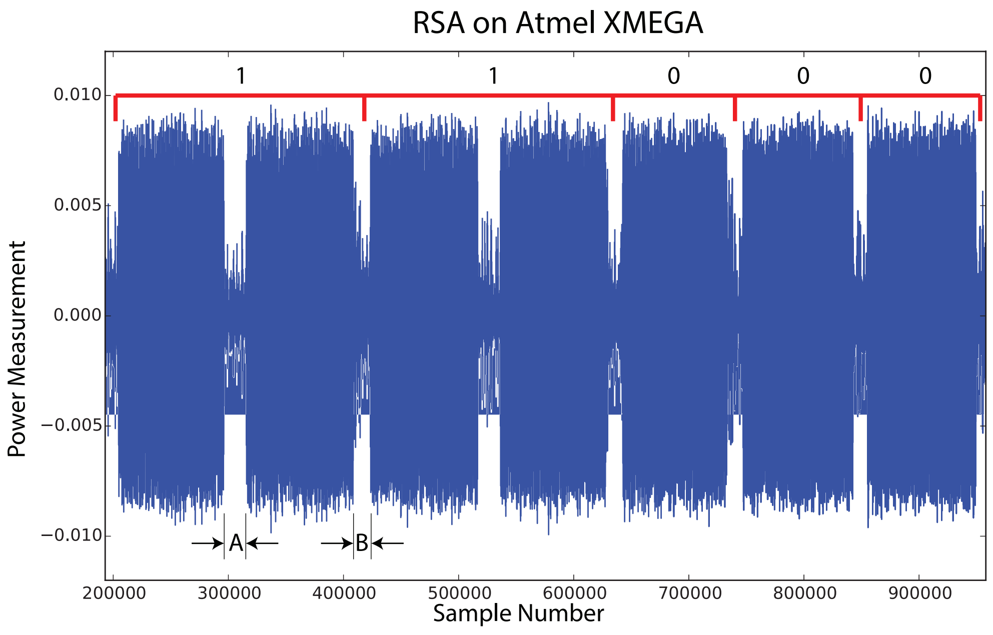


Figure 1: This exploits the data-path dependent code from Listing 1, which allows us to read the secret data off bit-by-bit.

1.2 Differential Power Analysis

The initial attack presented in [6] caused a digital device to execute an operation with both known and secret data. If we consider the case where that known and secret data is mixed together, we could define the known data as P, the secret data as K, and the operation as $C = f(P, K)$.

The DPA attack measures power consumption of the device during this operation. We can measure $i = 0, \cdots, N$ such operations with random known input data P_i, and constant unknown secret data K. We could set K to some assumed value K'. Assuming that K is a single byte, this presents 256 possibilities for the value of K'.

For each possibility of K', we could have a group of hypothetical outputs of the operation $C'_i = f(P_i, K')$ for each known input P_i, again where $i = 0, \cdots, N$. At this point we wish to determine which value of K' matches the true value of K on the hardware device running the algorithm.

One method presented in [6] is to target a single bit of the value of C' (and hence K'). For each hypothetical value of K' we can separate the power traces into two groups: one where a bit of C'_i is '1', one where the bit is '0'. If our hypothetical value of K' matched the true value K, we would expect a difference at some point in the mean power consumption between the two groups.

If our value was incorrect, we would expect no such difference, as the grouping could simply be considered as a random grouping of the traces into the two sets. In practice, such difference does exist when correctly grouped. Fig. 2 shows an example of the difference between the mean of two such groups, which have been correctly grouped into a set where the internal bit is '1' and the set where the internal bit is '0'. Note the trace shows us the location in time where the data is manipulated, as all other samples where the processor is *not* handling the data we targeted have the same mean.

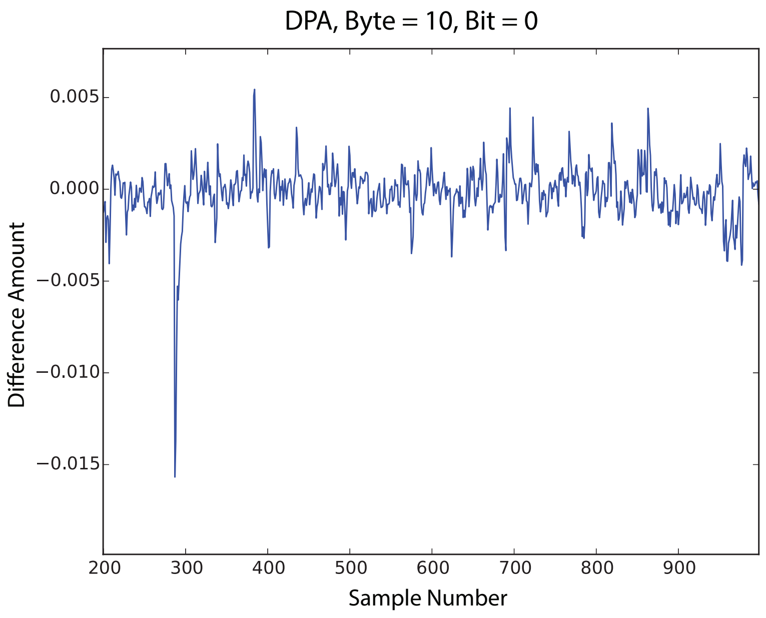


Figure 2: This demonstrates a DPA attack on a single bit, the large spike occurs at the instance in time where the processor is manipulating the data of interest.

This demonstrates that on a fundamental level devices do leak information regarding the state of the internal data bus. One additional consideration is how this can specifically be used to break cryptographic implementations, as it would appear this method still requires some level of "guess and check". This "guess and check" however is not performed over the entire key-space.

The implementation of cryptographic algorithms involves operations on individual bytes or words of data. For example although AES-256 involves a key of 256 bits (32 bytes), the "guess and check" for performing DPA only involves guessing a single bit at a time. This means a very tractable problem of performing $2^1 \times 256$ guesses, something even a typical personal computer can accomplish in a few seconds.

The hypothetical output of some function we are targeting is typically referred to as the "intermediate value", as we are targeting some value within the entire operation of the algorithm. When attacking AES this is often after the first round of the SubBytes operation, as the non-linear property of the SubBytes improves our attack by eliminating the linear relationship between the input and intermediate values. In addition one byte of the plaintext will directly mix with one byte of the secret key, reducing the complexity of performing the guess and check operation.

While the DPA attack was the first proposed methods, more efficient methods of discovering the secret key information using the power traces exist. We'll discuss two major methods next: the Correlation Power Analysis (CPA) attack and template attacks.

1.2.1 Correlation Power Analysis (CPA)

Whereas DPA looked at simple differences between two groups of data, the CPA attack develops more precise assumptions on the power consumption and relation on an internal data bus. The CPA attack was first present in 2004 by Brier et al. in [3], and will be summarised here.

For a simple 8-bit microcontroller, we can use a "leakage model" that suggests the instantaneous power consumption is related to the number of bits set to '1' on the internal databus. This assumption is based on two factors: (1) our previous knowledge that moving the state of a line takes a certain amount of power, and (2) knowledge that microcontrollers set their internal buses to a constant state before the final value is loaded.

This constant state is known as the 'precharge' state. This precharge has been used since the early design of microcontrollers, where it was easier and faster to design a bus with precharge logic to pull the bus to the '1' state, requiring each module driving the bus to only have the pull-down logic (rather than requiring full push-pull and enable transistors on each bus connection)[7].

More recent devices may pre-charge to other levels, such as precharging to a level between '1' and '0', with the objective being to reduce the power and time required to transition to the final level [4]. This pre-charge would require push-pull drivers at each bus connection, so is targeting improved performance rather than a simplified design.

From an attack perspective, specifics of the pre-charge are irrelevant. Instead the attacker cares there is a constant starting level, meaning a linear relationship between the number of bits set to '1' on the databus and the power consumption. Depending on the precharge level and measurement style this relationship may have a positive or negative slope. Without this pre-charge we instead have a relationship between the *change* in bits between two bus states, and thus would also need to know (or guess) the previous state.

The case of the pre-charge will be referred to as the Hamming Weight (HW) model, where leakage is assumed to be related to the number of bits set to '1' on the bus. Without the precharge we would have the Hamming Distance (HD) model, where the leakage is related to the number of bits changing states on the bus.

As a validation of this previous work, we measured the power consumption of an 8-bit microcontroller (Atmel ATMega328P) at the moment it is manipulating data with various number of bits set to '1'. The results in Fig. 3 show an excellent relationship between the HW of the data and the power measurement.

The basic equation for a CPA attack, where $r_{i,j}$ is the correlation coefficient at point j for hypothesis i, the actual power measurement is $\vec{t_{d,j}}$ of trace number d at point j, and $p_{d,i}$ is the hypothetical power consumption of hypothesis i for trace number d, with a total of D traces is given in equation (1). This equation is simply an application of the Pearson's correlation coefficient given in equation (2), where $X = \vec{p}$, and $Y = \vec{t}$.

$$r_{i,j} = \frac{\sum_{d=1}^{D} \left[\left(p_{d,i} - \overline{P_j}\right) \left(t_{d,j} - \overline{t_j}\right) \right]}{\sqrt{\sum_{d=1}^{D} \left(p_{d,i} - \overline{P_j}\right)^2 \sum_{d=1}^{D} \left(t_{d,j} - \overline{t_j}\right)^2}} \tag{1}$$

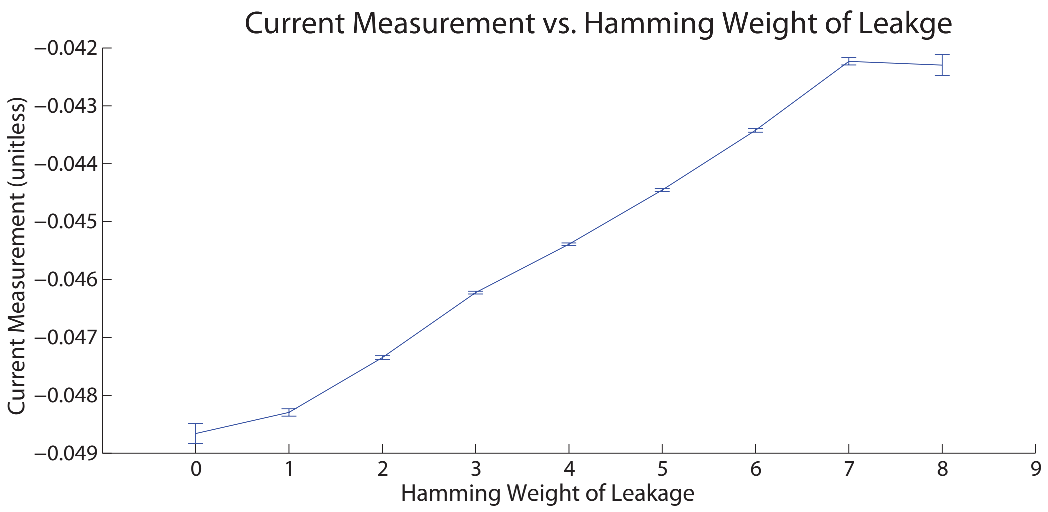


Figure 3: Power consumption of device under attack performing an operation on data with different Hamming Weights (HW), showing the average current consumption of the AtMega328P microcontroller for each possible hamming weight of an 8-bit number. Error bars show 95% confidence on average (based on the sample standard deviation).

$$\rho_{X,Y} = \frac{cov\left(X,Y\right)}{\sigma_X \sigma_Y} = \frac{E\left[\left(X-\mu_X\right)\left(Y-\mu_Y\right)\right]}{\sqrt{E\left[\left(X-\mu_X\right)^2\right]}\sqrt{E\left[\left(Y-\mu_Y\right)^2\right]}} \tag{2}$$

The form given in these equations is referred to as the *normalized cross-correlation*, and frequently used in image processing applications for matching known templates to an image.

2 It's Time For Statistics

Power traces leak some information about the data being processed on a device. With this in mind, our next goal is to find whether there is enough leakage to perform a successful side channel attack. Simply looking at the power traces is not enough: it is difficult to tell whether a power difference contained useful information or if it was just an (un)lucky streak. To do this, we need some help from statistics.

2.1 Analyzing DPA Leakage

The simpler case is DPA. In a DPA attack, we put our power traces into two groups by looking at one bit of C_i': one group where this bit is 0, and another group where it's 1. Then, our goal is to check if these two groups are significantly different at any point in time. In terms of statistics, we have a null hypothesis that the average power consumption is the same in both groups:

$$H_0 : \mu_0 = \mu_1 \tag{3}$$

Then, if we can reject this hypothesis, that means that we've found a significant difference between the two groups – enough of a difference for a DPA attack.

To try to reject the null hypothesis, we use a Student's t-test. The t statistic for one point in time is

$$t = \frac{\overline{P}_0 - \overline{P}_1}{s/\sqrt{N}} \tag{4}$$

Here, $\overline{P}_0$ and $\overline{P}_1$ are the average power measurements in the two groups, s is the standard deviation of all of the measurements, and N is the total number of power traces. If t is small, then the two groups do not look significantly different. However, if t is large, then we can reject the null hypothesis: there is a significant, consistent difference between the groups.

We need to pick a threshold to distinguish between "small" and "large" values of t. This threshold determines our false positive rate. For example, if $t = 2$, then there is a 5% chance that the null hypothesis is true. This chance drops to 0.2% when $t = 3$, and to 0.006% when $t = 4$. Using a higher threshold causes less false positives, but might miss some power differences that are still useful in practice.

2.2 Analyzing CPA Leakage

T-tests can only check for differences between two groups, and CPA attacks look at more than two groups. In the example from the last section, the Hamming weight of a 1-byte value can range from 0 to 8 – 9 different values. To deal with the large number of groups, we can use linear regression. Looking at the data in Figure 3, we can use least-squares regression to fit the line

$$P = b_0 + b_1 \cdot \mathrm{HW} \tag{5}$$

where P is the power consumption and HW is the Hamming weight of the data. The important part of this curve is b_1: if it is far from 0, then the data has a strong effect on the power.

To test whether b_1 is different from 0, we can calculate the standard error of this coefficient as

$$se(b_1) = \sqrt{\frac{s^2}{N \cdot \mathrm{var}(\mathrm{HW})}} \tag{6}$$

where s^2 is the variance of the residuals (the distances between the line and the data) and var(HW) is the variance of the Hamming weights. Then, the t statistic is

$$t = \frac{b_1}{se(b_1)} \tag{7}$$

If t is large, b_1 is unlikely to be 0. The interpretation of this value, such as the false positive rate for any t value, is the same as the regular t-test. This analysis method also handles DPA data by putting the two groups at HW $= 0$ and HW $= 1$.

2.3 Alternatives to T-Tests

There are two ways to improve these analyses: using Welch's t-test, and using more general tests such as ANOVAs.

The Student's t-test makes two significant assumptions: it requires the two groups to have the same number of datapoints and the same standard deviations. Welch's t-test is a better alternative when these assumptions don't hold. It deals with unequal group sizes and variances by making the $s/\sqrt{N}$ term slightly more complex. It is also equivalent to the Student's t-test when these assumptions do hold. However, there is no simple analogue for Welch's t-test with many groups.

The CPA leakage analysis assumes that there is a linear relationship between the data's Hamming weight and the power consumption. This model of leakage is not always accurate. Our example leakage data (Figure 3 shows that the points with a Hamming weight of 0 or 8 do not follow the same trend as the other weights. A more general way to look for leakage is to check if the data has any effect on the power consumption – linear or not. To do this, the normalized inter-class variance (NICV) [2] is

$$\mathrm{NICV} = \frac{\mathrm{var}(\mathbb{E}(P|HW))}{\mathrm{var}(P)} \tag{8}$$

This test asks: if we know the Hamming weight of the data, does this give us a more accurate picture of the power consumption? This test is more difficult to convert to a practical CPA attack, so we leave it for future work.

3 ChipWhisperer-Lint Automated Leakage Model

Previous leakage testing methods have come up with extensive test plans with specific knowledge about one encryption algorithm. For example, test vector leakage assessment (TVLA) for AES uses a battery of tests that assume side channel leakage will only occur in specific parts of the algorithm [5, 1].

ChipWhisperer-Lint does not claim to find any specific leakage models based on hardware knowledge. Instead it looks at the most likely locations items could be stored in registers, and from that "brute-forces" the potential leakage models. This was inspired by papers demonstrating non-obvious leakage models, with our objective of determining if there are other odd leakage models that may be useful in devices[8].

For example considering the AES algorithm, we could write out the following states of the algorithm:

```
Input: Plaintext
Input: Key
Round 0: AddRoundKey Output (Key \oplus Plaintext)
Round 1: Sub-bytes Input
Round 1: Sub-bytes Output
Round 1: Shift-rows Input
Round 1: Shift-rows Output
...
Output: Ciphertext
```

After each of these operations we know what is held in a CPU register or memory location. We have this knowledge because we are NOT an attacker, but an evaluator/tester. That is we have complete knowledge of the encryption key and input to the algorithm, so could recover every specific state.

We know that power can be related to the Hamming weight of data in a register, or the Hamming distance between the two values present in the register as one gets overwritten.

Thus we can simply combine every possible Hamming weight (HW) of the intermediate registers, along with the Hamming distance (HD) of potential ways states could get overwritten.

Our short list above could become the following possible leakages:

```
HW: Input: Plaintext
HW: Input: Key
HW: Round 0: AddRoundKey Output (Key XOR Plaintext)
HW: Round 1: Sub-bytes Input
HW: Round 1: Sub-bytes Output
HW: Round 1: Shift-rows Input
HW: Round 1: Shift-rows Output
...
```

```
HD: Input Plaintext TO Input Key
HD: Input Plaintext TO Round 0: AddRoundKey Output (Key XOR Plaintext)
HD: Input Plaintext TO Round 1: Sub-bytes Input
HD: Input Plaintext TO Round 1: Sub-bytes Output
HD: Input Plaintext TO Round 1: Shift-rows Input
HD: Input Plaintext TO Round 1: Shift-rows Output
...
HD: Input Plaintext TO Output: Ciphertext
...
HD: Round 1: Sub-bytes Input to Round 1: Sub-bytes Output
...
```

A few notes about this list:

- Several of these are identical - the AddRoundKey Output and Sub-bytes Input are the same thing. Thus those steps should be collapsed into one to reduce the combinational growth.
- Some of the leakage models might seem impossible or nonsense. We do not filter these, as a number of odd leakage models have been found previously [8].
- The described method is looking at bytes in same location – i.e., looking at byte 0 or word 0 to determine the overwritten Hamming distance. If a single S-Box is used you might see leakage between the Hamming distance of the S-Box input (see XMEGA), so a method to account for shifts between states is also needed.

In addition to some of the leakage models seeming to be nonsense, more importantly some of the combinations end up being identical. For example `HD: Input Plaintext TO Round 0: AddRoundKey Output (Key XOR Plaintext)` is identical to `HW: Input Key`, since this is effectively performing the operation $Plaintext \oplus (Key \oplus Plaintext)$.

These are currently not filtered out either. The resulting leakage numbers will be identical, making it easy to identify the cloned values. ChipWhisperer-Lint has some ability to define leakages which are ignored (such as the Hamming weight of the plaintext) via the configuration file.

3.1 Output Format

ChipWhisperer-Lint currently uses either Welch's T-Test (when testing between two groups) or the n-group Student T-Test. The leakage information for the n-group Student T-Test is typically used as a potential leakge function for a CPA attack, which is then used to validate if a attackable leakage exists.

Compared to the TVLA test that is performing pass/fail testing[5], we are attempting to determine likely leakage models that require better study. As such we use a smaller threshold that is more likely to generate false positive, typically we are using a value of 3.0. The potential leakage models are then validate with attacks – the "false positive" problem is less pronounced, since additional validation of any leakages is performed.

The output is a plot of T-statistic vs. sample number, with any T-statics reaching above the threshold being marked as a “failure”. Note the tool marks “FAIL” or “PASS” based only on the information regarding the T-Statistic – additional validation is required to confirm potential leakage models result in secret key material leakage.

The tool also has a mode to perform validation per classic TVLA testing, including random vs. fixed testing and other specific leakage modes. Using these modes allows ChipWhisperer-Lint to perform true pass/fail testing, per the TVLA proposal[5], later codified in ISO/IEC 17825.

3.2 Configuration

ChipWhisperer-Lint is run based on two inputs: a trace data file (in ChipWhisperer project file format), and a Configuration file. The configuration file defines what groups to compare (i.e., what leakages to look for).

Part of an example configuration file is show below:

```
HD: Plaintext to Round 3: ShiftRows Output
Plaintext
FF000000000000000000000000000000
Round 3: ShiftRows Output
FF000000000000000000000000000000
N
```

Note the file contains the two leakage states, where the mask showing the `FF00...00` is showing the data required to be XORd together for the leakage. This example would demonstrate a Hamming distance (HD) leakage.

4 Test Bench Setup

The capture required for the power analysis is performed with the low-cost and open-source ChipWhisperer-Lite Capture (P/N NAE-CWLITE-CAPTURE) along with a UFO Target Board (P/N NAE-CW308). The UFO board[3] allows mounting a target board, which are custom-designed boards with the target chip. This allows easy comparison of the various devices under similar conditions.

The test setup is shown in Figure 4. The ChipWhisperer-Capture performs synchronous capture of the target device power trace. This synchronous capture is unique to the ChipWhisperer platform, as it means that captured data is directly related to clock cycles of the target device. While a regular oscilloscope can be used, a much faster sample rate is required compared to the ChipWhisperer capture platform[10, 11].

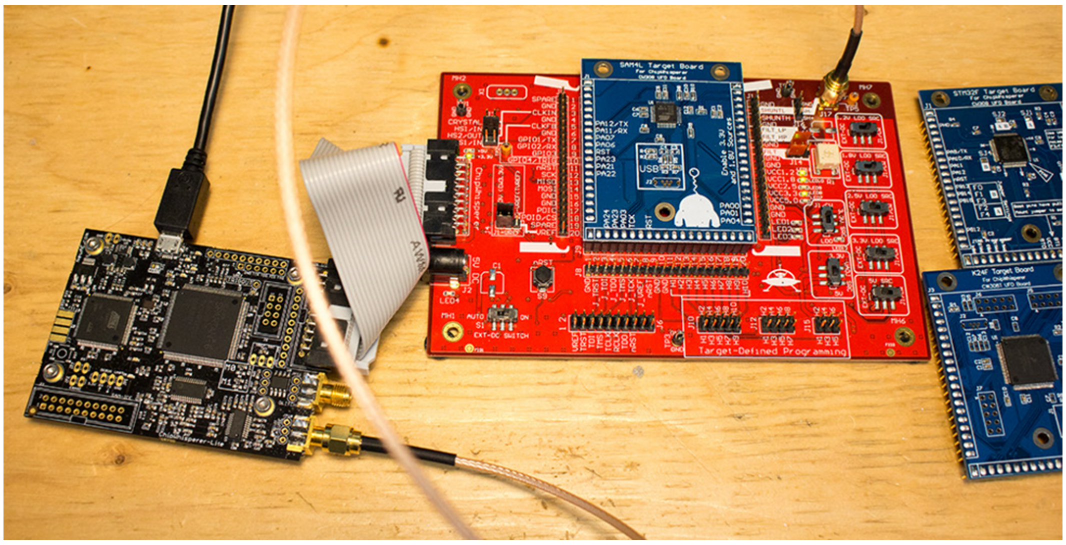


Figure 4: ChipWhisperer-Lite Capture is used with a UFO Target Board, allowing swapping of various target boards onto the device.

[3]The UFO Board is named as such because it is used for probing things.

5 Software Cryptography

While ChipWhisperer-Lint was designed to assist with finding leakage models for hardware cryptography, we have also pointed it towards some software stacks to provide a ”bounds” on the security level.

In addition, ChipWhisperer-Lint can be useful for finding the interaction between software and hardware. For example code compiled on one platform may be more leaky with side-channel power analysis compared to code on other platforms.

5.1 MBED TLS

As an example of what a “software” leakage looks like, we have evaluated MBED TLS running on the STM32F415 device. This is running AES encryptions at 7.37 MHz. This implementation uses T-Tables and represents a reasonably recent/powerful implementation of AES you might run on an Arm device. The waveform of the encryption is shown in Figure 5. The leakage model is the standard Hamming weight model using the SubBytes output from [3].

It should be noted that power analysis recovers the key in about 200 traces here, despite the leakage model not perfectly matching the algorithm. The S-Box itself does not appear in the code, instead it is encoded into the T-Table implementation used by MBED-TLS. But since the value IS encoded inside this table (even if not directly visible), the correlation peak as in 6 still shows a strong peak at one location.

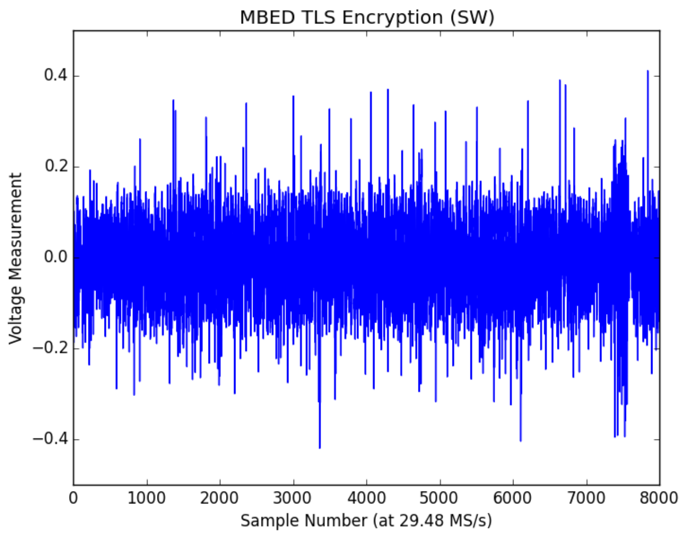


Figure 5: MBED-TLS waveform during encryption operation.

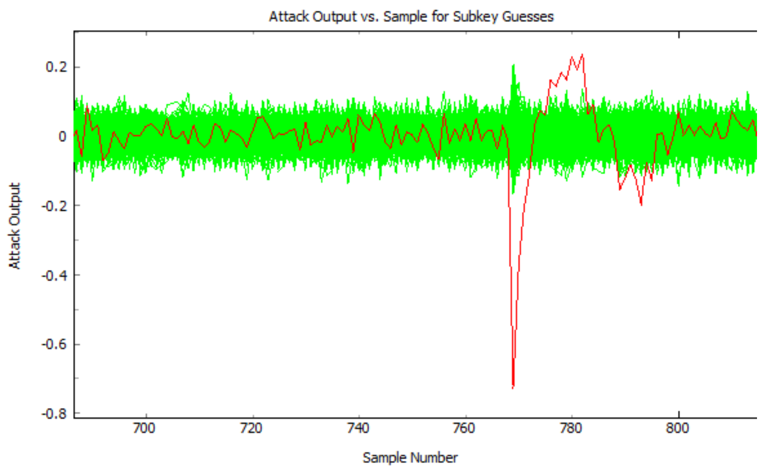


Figure 6: MBED-TLS Shows strong correlation of the Hamming weight leakage at the S-Box output.

5.2 ROM Implementations on Si4010

The Silicon Labs Si4010 is an 8051-core microcontroller, targeting small remote control applications such as garage door openers. This device contains an "AES accelerator" (per the feature section of the datasheet). In reality, the AES accelerator actually implements only a few specific operations such as the S-Box lookup.

The rest of the AES algorithm is performed in software. This software is ROM code, which increases the available FLASH memory the end user can dedicate to their application. The user can thus call an AES encryption algorithm, and be provided with the results.

As expected, the leakage is very similar to a software implementation. The critical leaking portion (the S-Box Hamming weight) is identical to the MBED TLS software example. This device can be broken with 40-50 encryption traces using the CPA attack.

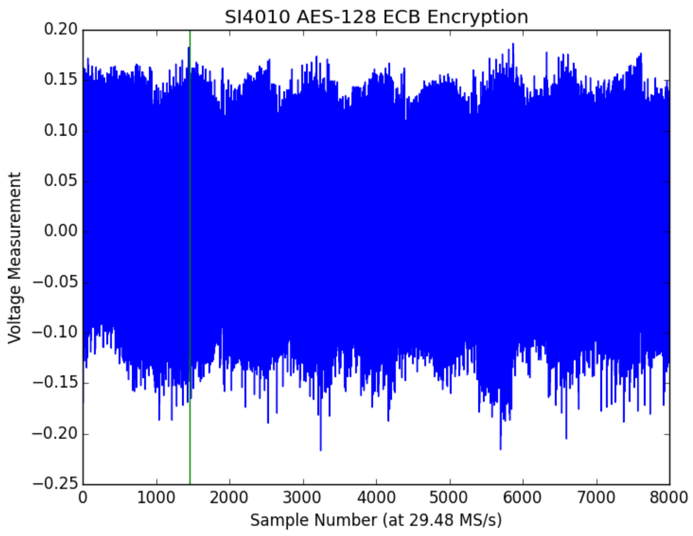
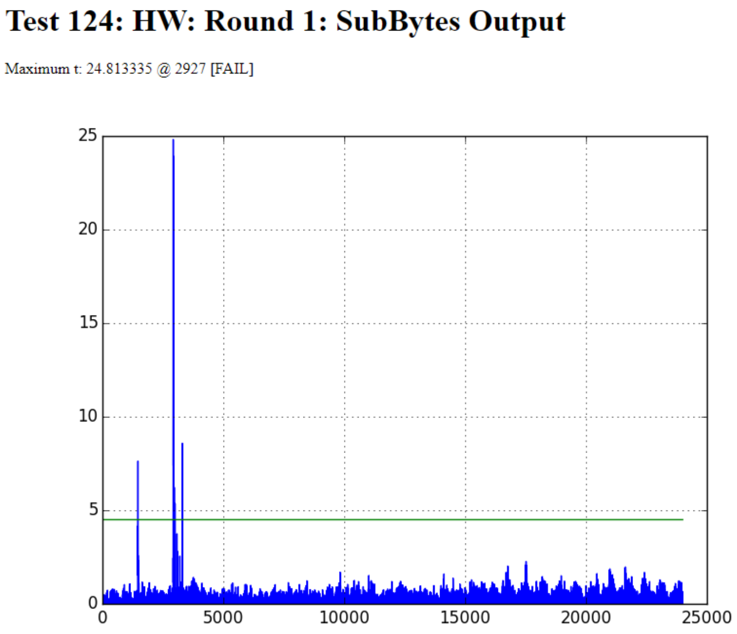


Figure 7: Si4010 AES ROM code. Green and Red bars indicate start and end of encryption.


Test 124: HW: Round 1: SubBytes Output

Maximum t: 24.813335 @ 2927 [FAIL]

Figure 8: Si4010 AES showing Hamming weight leakage.

Table 1: Si4010 Capture Parameters

Clock	Internal clock, as the microcontroller cannot be switched off the internal oscillator. An external oscillilator input is available, but used only to calibrate the internal oscillator.
Shunt	10 Ohms.
Capture Hardware	ChipWhisperer-Lite Capture.
Sample Rate	29.4 MS/s (asynchronous to device clock).

5.3 Leakage due to calling functions

One source of potential leakage is when the function calling a hardware cryptographic block contains side-channel power analysis leakages. This is particularly a problem when evaluating a hardware crytographic function itself, since the test functions could be introducing leakage that do not exist in the actual product.

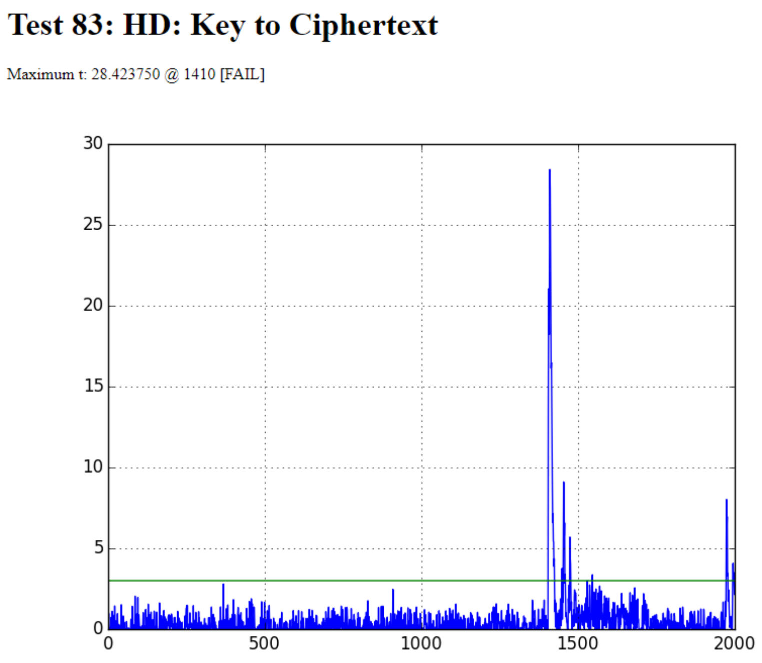
Consider a perfectly secure AES hardware block, which has two register banks. The device is a 32-bit processor, with four key and four input data registers. The four key and input data registers make up the 128-bit AES input size. We will call the key registers REG_KEY0, REG_KEY1, ..., REG_KEY3, and the data input register REG_DATA0, ..., REG_DATA3.

A reasonable method of calling this might be as in the listing below:

```
void encrypt_data_block(uint8_t * data, uint8_t * key)
{
    uint32_t temp1, temp2, temp3, temp4;

    temp1 = *((uint32_t *)data + 0);
    temp2 = *((uint32_t *)data + 1);
    temp3 = *((uint32_t *)data + 2);
    temp4 = *((uint32_t *)data + 3);

    REG_KEY0 = temp1;
    REG_KEY1 = temp2;
    REG_KEY2 = temp3;
    REG_KEY3 = temp4;

    temp1 = *((uint32_t *)key + 0);
    temp2 = *((uint32_t *)key + 1);
    temp3 = *((uint32_t *)key + 2);
    temp4 = *((uint32_t *)key + 3);

    REG_DATA0 = temp1;
    REG_DATA1 = temp2;
    REG_DATA2 = temp3;
    REG_DATA3 = temp4;
}
```

Unfortunately, you have just introduced a strong side-channel leakage. If you compare the value stored in register `temp1`, you will notice it first has the value of the first 32-bits of data, then the first 32-bits of key.

The step when the key replaces the data will give you a Hamming distance leakage at that point in the code. This can be seen in some of the ChipWhisperer-Lint test results. It is most obvious when very odd leakages come up - for example leakage between the Cipher-Text and Encryption Key is shown in Figure 9.



Test 83: HD: Key to Ciphertext

Maximum t: 28.423750 @ 1410 [FAIL]

Figure 9: Leakage from the first-round key and ciphertext during an encryption operation is most likely overwriting of the key storage variable with the output ciphertext during the function execution.

This leakage is not a true leakage from the encryption core, but an artifact of register re-use. This particular leakage was obviously bogus, since the leakage was so strong and there was no reasonable physical explanation for it.

More commonly, this leakage appears as the Hamming distance between the key and plaintext, one should not immediately assume they have successful broken the output of AddRoundKey (KEY $\oplus$ PLAINTEXT), but instead may have introduced the leakage themselves. The register re-use leakage should be suspected when there is a word-specific or byte-specific offset in the leakage location. That is the first 32-bits have leakage at a specific point, but the next 32-bits have leakage at a different point. Such time steps suggest a software leakage, as the XOR operation in hardware would likely be completed as one step.

This can easily be confirmed by ensuring the key is not set on every use, assuming the core does not require the key set before each use. When performing CPA attacks on the hardware cores, we have always disabled the ”key set” function to ensure there were no unexpected results.

Chips which provide ROM functions for interfacing with the hardware peripheral need to be especially careful of this leakage. While the ROM function enforces specific usage of the peripheral, there is a risk that any leakage introduced by the ROM function cannot be fixed.

6 Regression Testing

An interesting use-case for automated TVLA (or other side-channel) testing is performing regression testing to detect issues arising in code.

As mentioned, incorrectly overwriting a sensitive value with an attacker-controlled (or attacker-known) value is a good way to introduce a power analysis leakage. Changes in compilers may introduce such leakage, even if it was not initially contained within the code.

This can be extended to build across different device architectures to find leakage resulting from interaction between the compiler and hardware. This works by building a wide variety of firmware images across different compilers and for different devices. The firmware images are programmed onto the devices, and the power analysis measurements performed. The setup is shown in Figure 10. An example of the results is shown in Figure 11. Usage of ChipWhisperer allows a large hardware test bed to be built with widely available commodity hardware.

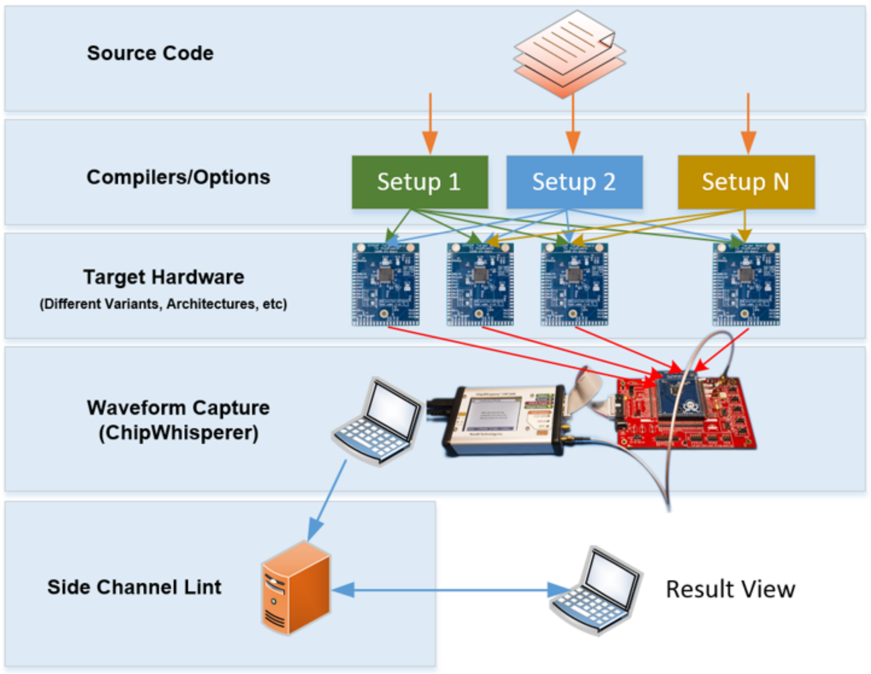
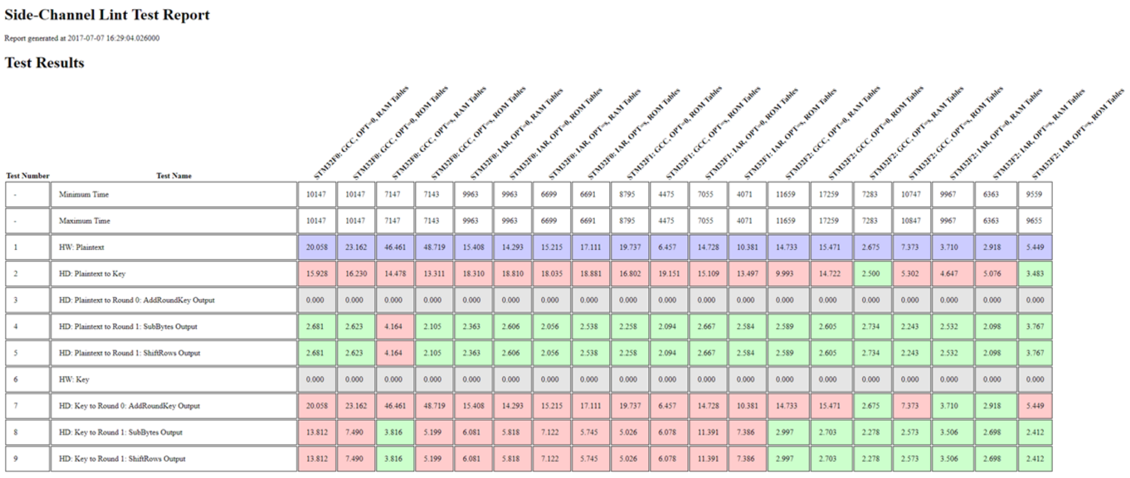


Figure 10: ChipWhisperer-Lint can be used to perform robust software validation across multiple devices.



Side-Channel Lint Test Report

Report generated at 2017-07-07 16:29:04.026000

Test Results

Test Number	Test Name	STM32F0: GCC, OPT=0, RAM Tables	STM32F0: GCC, OPT=0, ROM Tables	STM32F0: GCC, OPT=s, RAM Tables	STM32F0: GCC, OPT=s, ROM Tables	STM32F0: IAR, OPT=0, RAM Tables	STM32F0: IAR, OPT=0, ROM Tables	STM32F0: IAR, OPT=s, RAM Tables	STM32F0: IAR, OPT=s, ROM Tables	STM32F1: GCC, OPT=0, ROM Tables	STM32F1: GCC, OPT=s, ROM Tables
-	Minimum Time	10147	10147	7147	7143	9963	9963	6699	6691	8795	4475
-	Maximum Time	10147	10147	7147	7143	9963	9963	6699	6691	8795	4475
1	HW: Plaintext	20.058	23.162	46.461	48.719	15.408	14.293	15.215	17.111	19.737	6.457
2	HD: Plaintext to Key	15.928	16.230	14.478	13.311	18.310	18.810	18.035	18.881	16.802	19.151
3	HD: Plaintext to Round 0: AddRoundKey Output	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4	HD: Plaintext to Round 1: SubBytes Output	2.681	2.623	4.164	2.105	2.363	2.606	2.056	2.538	2.258	2.094
5	HD: Plaintext to Round 1: ShiftRows Output	2.681	2.623	4.164	2.105	2.363	2.606	2.056	2.538	2.258	2.094
6	HW: Key	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
7	HD: Key to Round 0: AddRoundKey Output	20.058	23.162	46.461	48.719	15.408	14.293	15.215	17.111	19.737	6.457
8	HD: Key to Round 1: SubBytes Output	13.812	7.490	3.816	5.199	6.081	5.818	7.122	5.745	5.026	6.078
9	HD: Key to Round 1: ShiftRows Output	13.812	7.490	3.816	5.199	6.081	5.818	7.122	5.745	5.026	6.078

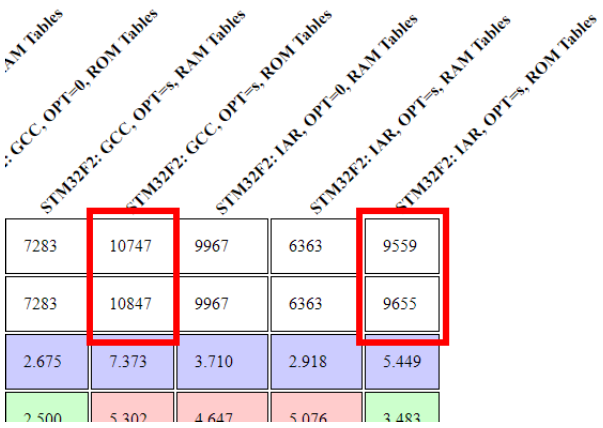
Test Number	Test Name	STM32F1: IAR, OPT=0, ROM Tables	STM32F1: IAR, OPT=s, ROM Tables	STM32F2: GCC, OPT=0, RAM Tables	STM32F2: GCC, OPT=0, ROM Tables	STM32F2: GCC, OPT=s, RAM Tables	STM32F2: GCC, OPT=s, ROM Tables	STM32F2: IAR, OPT=0, RAM Tables	STM32F2: IAR, OPT=s, RAM Tables	STM32F2: IAR, OPT=s, ROM Tables
-	Minimum Time	7055	4071	11659	17259	7283	10747	9967	6363	9559
-	Maximum Time	7055	4071	11659	17259	7283	10847	9967	6363	9655
1	HW: Plaintext	14.728	10.381	14.733	15.471	2.675	7.373	3.710	2.918	5.449
2	HD: Plaintext to Key	15.109	13.497	9.993	14.722	2.500	5.302	4.647	5.076	3.483
3	HD: Plaintext to Round 0: AddRoundKey Output	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4	HD: Plaintext to Round 1: SubBytes Output	2.667	2.584	2.589	2.605	2.734	2.243	2.532	2.098	3.767
5	HD: Plaintext to Round 1: ShiftRows Output	2.667	2.584	2.589	2.605	2.734	2.243	2.532	2.098	3.767
6	HW: Key	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
7	HD: Key to Round 0: AddRoundKey Output	14.728	10.381	14.733	15.471	2.675	7.373	3.710	2.918	5.449
8	HD: Key to Round 1: SubBytes Output	11.391	7.386	2.997	2.703	2.278	2.573	3.506	2.698	2.412
9	HD: Key to Round 1: ShiftRows Output	11.391	7.386	2.997	2.703	2.278	2.573	3.506	2.698	2.412

Figure 11: Non-obvious leakage may result from combinations of compilers, compiler flags, and hardware targets.

This summary page can be useful to understand where leakage exists due to software optimization or changes. A threshold is used to decide what leakage we care about, and leakages that have been previous validated can be ignored. This workflow is similar to analysis tools such as PCLint (and hence the name of ChipWhisperer-Lint).

For example, building MBLED-TLS AES on the STM32F2 series results in non-constant execution time. This is shown in Figure 12. This non-constant time appears like a cache timing attack, but there is no cache enabled on this device build.

Instead, the non-constant timing is actually due to the ”flash accelerator” in these devices. The flash accelerator performs 128-bit read from FLASH. Accessing an element inside that same 128-bit area is slightly faster than accessing one outside of it. Thus when performing the S-Box table lookup, we see non-constant timing depending on the sequence of lookups performed. It would be impossible to detect this with purely simulated test harnesses, since the effect is purely one that exists at the physical level.



AM Tables

: GCC, OPT=0, ROM Tables

STM32F2: GCC, OPT=s, RAM Tables	STM32F2: GCC, OPT=s, ROM Tables	STM32F2: IAR, OPT=0, RAM Tables	STM32F2: IAR, OPT=s, RAM Tables	STM32F2: IAR, OPT=s, ROM Tables
7283	10747	9967	6363	9559
7283	10847	9967	6363	9655
2.675	7.373	3.710	2.918	5.449
2.500	5.302	4.647	5.076	3.483

Figure 12: Non-constant execution time on the STM32F2 due to the flash accelerator.

7 Hardware Accelerators

The following looks at four examples of hardware accelerators built into microcontrollers. These were selected as a reasonable sample of microcontrollers that are on the market, which do not target specific security levels. That is there is no common criteria rating or similar available for them.

7.1 Leakage Models in Use

Of the possible leakage locations, three leakage models are used in breaking the devices. A short summary of the leakage models will be presented here.

First, it is worth remembering what AES looks like. A diagram of it is presented[4] in Figure 13. We often see a Hamming distance (number of bit changes) difference between the AES state. This is most easily exploited on the last round, as there is no MixColumn operation that would complicate our life, and require the learned authors to actually understand something about how Galois Field multiplication works rather than simply typing the words.

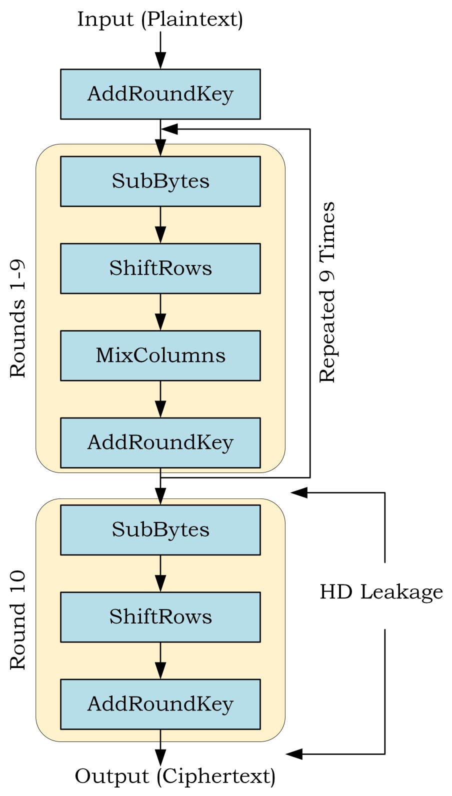


Figure 13: The AES-128 algorithm, with one common leakage source diagrammed.

[4]This figure is a tribute to the most ripped-off AES diagram in existence.

The first is the "state to state" leakage model. This leakage comes from overwriting of the AES state register, and is very common in hardware AES implementations that perform an entire round per clock cycle. There are many variations of AES implementations – the authors invite you to see how many papers with the theme "An efficient implementation of AES" exist if you are curious. But first, the leakage model for this classic hardware implementation is below:

```
class LastroundStateDiff(AESLeakageHelper):
    name = 'HD: AES Last-Round State'
    def leakage(self, pt, ct, key, bnum):
        # HD Leakage of AES State between 9th and 10th Round
        st10 = ct[self.INVSHIFT_undo[bnum]]
        st9 = inv_sbox(ct[bnum] ^ key[bnum])
        return (st9 ^ st10)

    def processKnownKey(self, inpkey):
        return keyScheduleRounds(inpkey, 0, 10)
```

This leakage model is part of the ChipWhisperer analyzer Python program. The model returns a byte that will be turned into a leakage by counting the number of 1's. Hence the XOR of the last two states being used to determine the number of bit flips.

One alternative form in particular is common. Note that the ShiftRows step is cheap in hardware – it involves only swapping some byte orders, which has no real cost since it just changes how we route the wires but no gates, latches, etc.

We could thus build the algorithm as in Figure 14. In this example we have moved some functions around – ShiftRows occurs before AddRoundKey, which only works if we also shift the RoundKey in the same order (hence AddShiftedRoundKey). The result is a simplified last round, and no need for the AddRoundKey "outside" the first round. The additional ShiftRows at the input is just a reordering of the hardware wires.

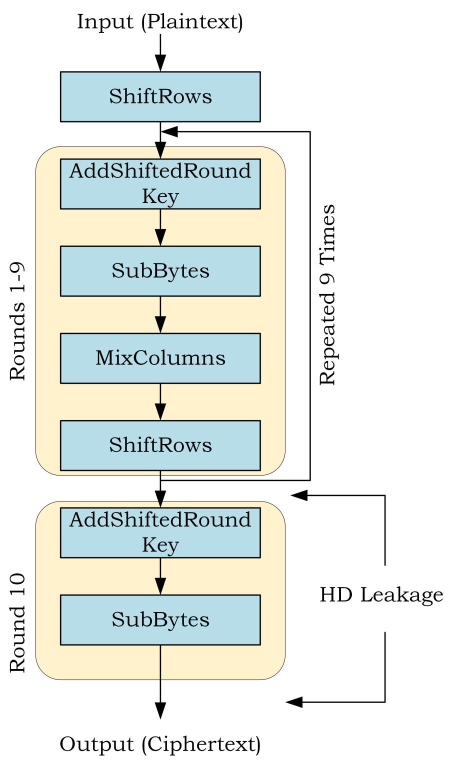


Figure 14: The AES-128 algorithm returns.

This modified algorithm requires the leakage model shown below:

```
class LastroundStateDiffAlternative(AESLeakageHelper):
    name = 'HD: AES Last-Round State Alternative'
    def leakage(self, pt, ct, key, bnum):
        st10 = ct[bnum]
        st9 = inv_sbox(ct[bnum] ^ key[bnum])
        return (st9 ^ st10)

    def processKnownKey(self, inpkey):
        k = keyScheduleRounds(inpkey, 0, 10)
        return self.shiftrows(k)
```

These operations assume the entire round occurs on a single clock cycle. For various reasons implementations may not perform a single round on a clock cycle (speed vs. area trade-offs mainly). One potential leakage source is the SubByte operation, and we could look for the Hamming distance of data from the input to output for this:

```
class SBoxInOutDiff(AESLeakageHelper):
    name = 'HD: AES SBox Input to Output'
    def leakage(self, pt, ct, key, bnum):
        st1 = pt[bnum] ^ key[bnum]
        st2 = self.sbox(st1)
        return (st1 ^ st2)
```

With some defined leakage models, let's see which ones appear in the test results.

7.2 ST STM32F415

The STM32F415 is a Cortex M4 microcontroller. It contains a cryptographic hardware accelerator. Note this particular device has an internal V_{core} regulator that is permanently internally connected to the V_{core} pin. Thus one cannot use an external "clean" power supply to reduce measurement noise.

To perform the shunt based power measurement on these devices, the technique detailed in [12] was used. This technique is to simply insert a shunt resistor between the internal regulator output pin (used for an external capacitor) and the external capacitor. See Figure 15 for this specific example.

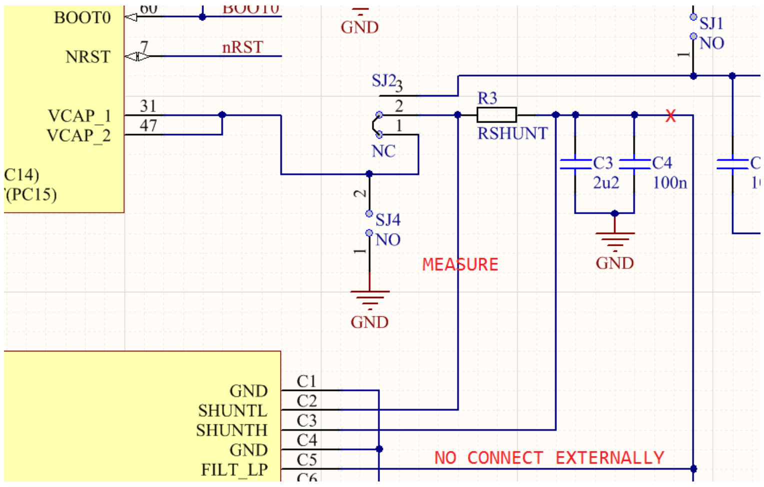


Figure 15: Power measurement is performed using the SHUNTL line. The SHUNTH / FILT_LP pins are not connected externally (done by removing the jumper on the CW308 UFO baseboard).

A complete encryption trace is given in Figure 16.

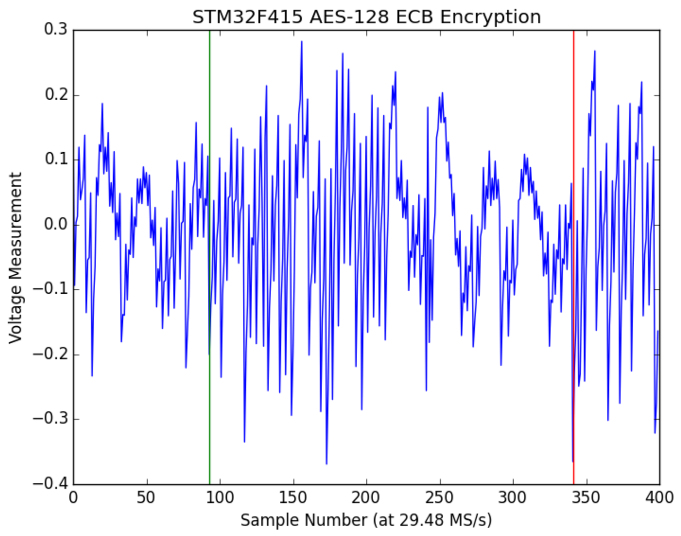


Figure 16: STM32F415 AES Hardware Peripheral. Green and Red bars indicate start and end of encryption.

Table 2: STM32F415 Capture Parameters

Clock	External 7.37 MHz clock.
Shunt	10 Ohms.
Capture Hardware	ChipWhisperer-Lite Capture.
Sample Rate	29.4 MS/s (synchronous to device clock).

7.3 NXP Kinetis K24F

The NXP Kinetis K24F provides a Cortex M4 core. This devices has an interesting leakage, as the strongest leakage appears to be a Hamming distance from the S-Box input to output. It is possible to recover the complete encryption key in approximately 14 000 traces.

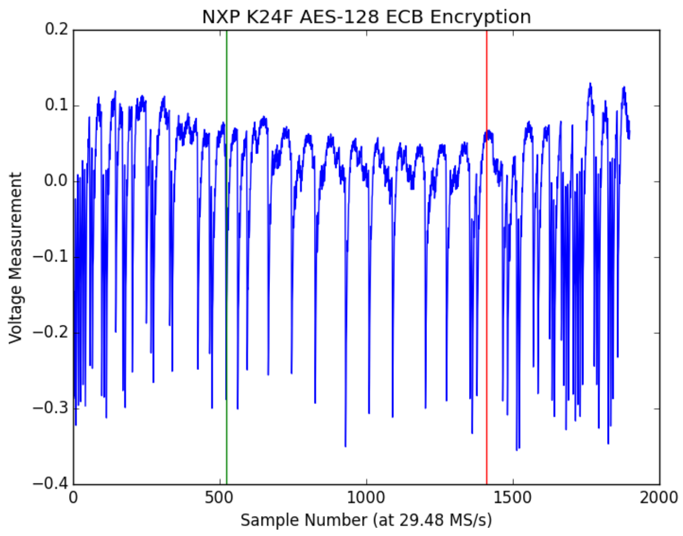
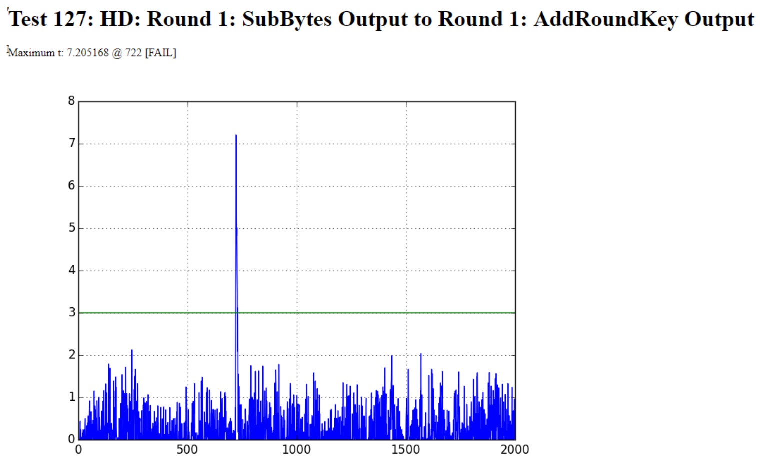


Figure 17: K24F AES Hardware Peripheral. Green and Red bars indicate start and end of encryption.

Table 3: K24F Capture Parameters

Clock	External 7.37 MHz clock.
Shunt	10 Ohms.
Capture Hardware	ChipWhisperer-Lite Capture.
Sample Rate	29.4 MS/s (synchronous to device clock).


Test 127: HD: Round 1: SubBytes Output to Round 1: AddRoundKey Output

Maximum t: 7.205168 @ 722 [FAIL]

Figure 18: The K24F Suggests S-Box Input to Output Leakage

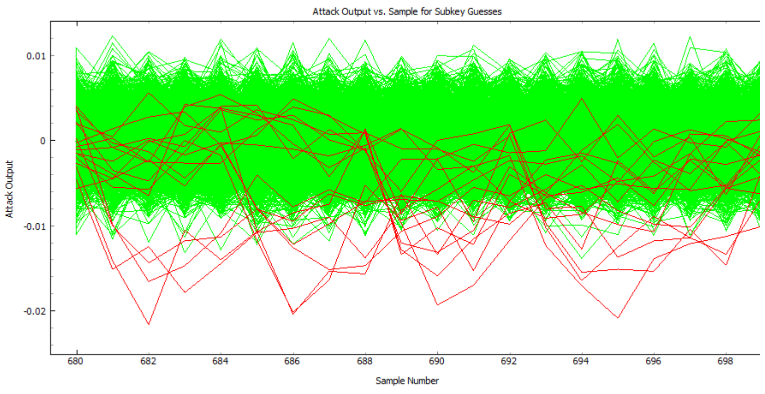


Figure 19: The K24F Leakage Suggests a Word-Wise Implementation (T-Table?)

7.4 Espressif ESP32

The Espressif ESP32 contains a hardware AES block, capable of performing AES-128 or AES-256. In addition to usage as a user peripheral for arbitrary data encryption, this peripheral is used as part of the real-time memory decryption, which allows an encrypted external SPI flash chip to be decrypted as different addresses are accessed.

We have only evaluated the peripheral when used in AES-ECB mode, as part of a user program. We make no claims about usage in memory decryption or other applications (at least until such analysis is performed).

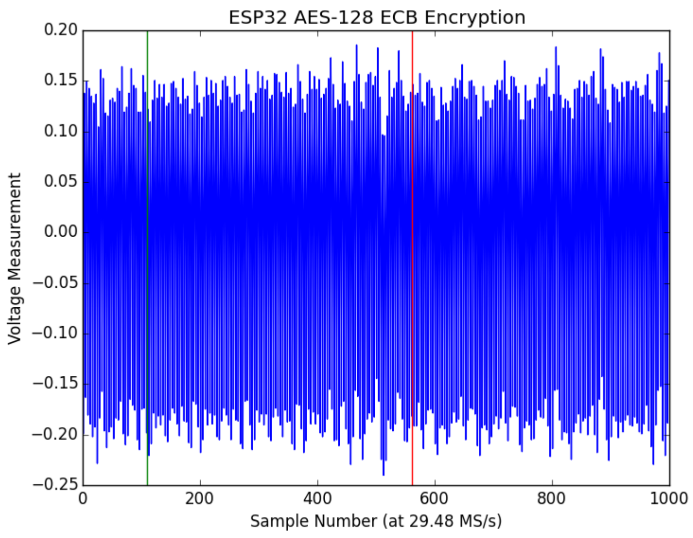


Figure 20: ESP32 AES Hardware Peripheral. Green and Red bars indicate start and end of encryption.

Table 4: ESP32 Capture Parameters

Clock	External 7.37 MHz clock.
Shunt	10 Ohms.
Capture Hardware	ChipWhisperer-Lite Capture.
Sample Rate	29.4 MS/s (synchronous to device clock).

The ESP32 provides several library functions for calling the AES routines. The exact code used is given below, which specifically calls a lower-level function (`ets_aes_cryp()`) than would normally be called by the user.

```
void aes128_enc(uint8_t* pt, uint8_t* k)
{
        esp_aes_context ctx;
        esp_aes_init(&ctx);
        esp_aes_setkey_enc(&ctx, k, 128);


        esp_aes_acquire_hardware();
        ets_aes_setkey_enc(ctx.enc.key, ctx.enc.aesbits);
        trigger_high();
        ets_aes_crypt(pt, pt);
        trigger_low();
        esp_aes_release_hardware();

}
```

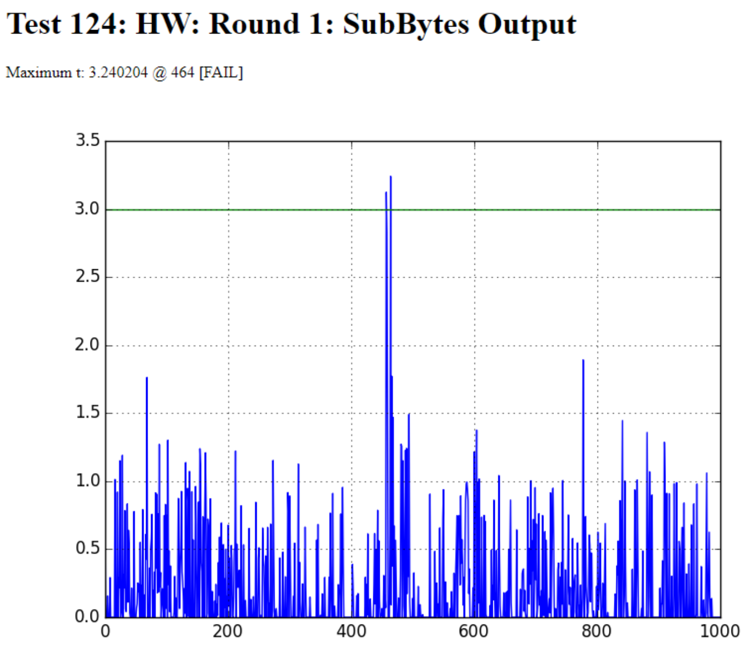


Figure 21: ESP32 AES CW-Lint Output suggests HW S-Box Leakage.

7.5 Microchip SAM4L

The Microchip SAM4L contains an AES core, which has several listed "countermeasures". The overall power trace is shown in Figure 22. Figure 23 shows a zoomed in portion of the power trace, where several traces with all countermeasures enabled are overlaid. The listed countermeasures (from the datasheet) are:

- Type 1: Randomly add one cycle to data processing
- Type 2: Randomly add one cycle to data processing (other version)
- Type 3: Add a random number of clock cycles to data processing, subject to a maximum of 11 clock cycles for key size of 128 bits
- Type 4: Add random spurious power consumption during data processing

When countermeasures 1-3 are all enabled there is noticeable desynchronization that stops a basic CPA attack, such that the attack will fail when countermeasures are turned on. The "add noise" (Type 4) countermeasure does not make a noticeable difference either visually or in attack results.

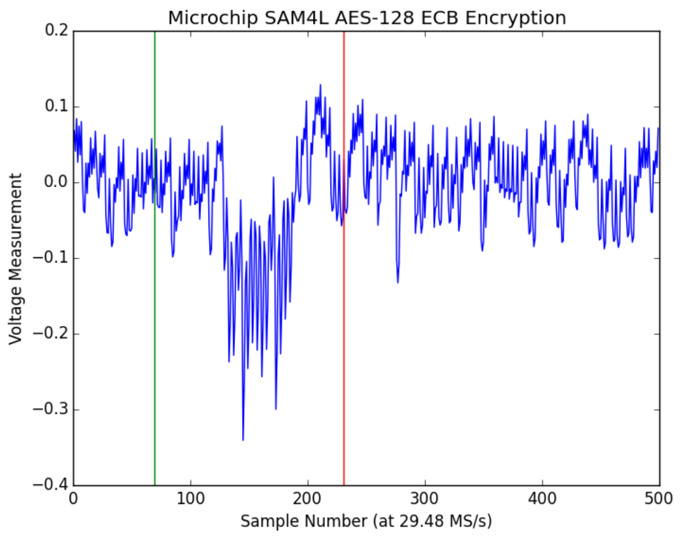


Figure 22: SAM4L AES Hardware Peripheral. Green and Red bars indicate start and end of encryption.

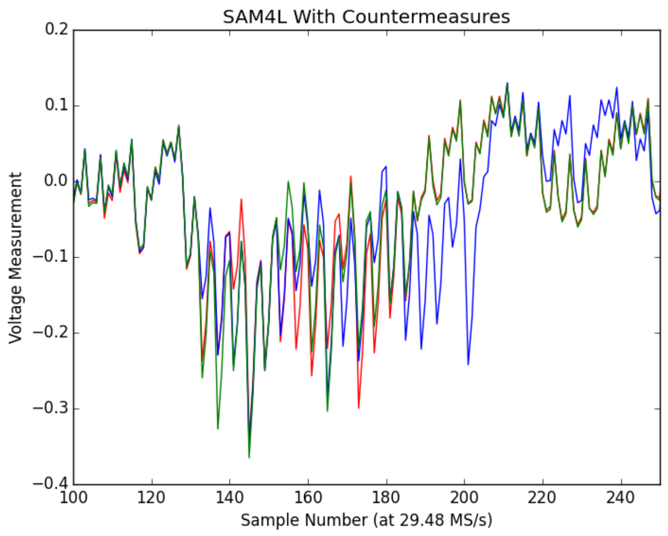


Figure 23: SAM4L Countermeasures add jitter within the algorithm.

Table 5: SAM4L Capture Parameters

Clock	External 7.37 MHz clock.
Shunt	10 Ohms.
Capture Hardware	ChipWhisperer-Lite Capture.
Sample Rate	29.4 MS/s (synchronous to device clock).

7.6 Summary

The following table shows a summary of the number of traces required for breaking the various microcontroller devices hardware AES implementation. The number of traces is comparable across the various devices, indicating a general security level that one could reasonable expected.

Note we have not fully evaluated the SAM4L with countermeasures enabled. Enabling countermeasures causes the CPA attack without any preprocessing to fail, tested up to 50 000 traces. Since we have not optimized the attack for another other target, we have not attempted to break the SAM4L using special preprocessing or filtering.

The 'AES-128 Cycles' includes the overhead of any calling libraries or ROM code. This presents an example of the time used in performing the encryption by the application code, and not the minimum or optimized time that is possible with any given device.

Table 6: Summary of CPA Attack Parameters and Results

Device	AES-128 Cycles	Leakage Model	Traces Req'd	Notes
STM32F415	493	Round to Round HD	6000	
Kinetis K24F	475	S-Box Input to Output HD	14 000	
ESP32	252	S-Box output HW	18 000	Calling ROM function directly
SAM4L	74	Round to Round HD	3000	Countermeasures DISABLED
SAM4L	81	?	?	Countermeasures ENABLED

8 Conclusion

ChipWhisperer-Lint is a tool for detecting side channel leakage models. While previous work has concentrated on general leakage detection, this tool has focused on moving from generic detection to specific leakage models. In particular, a brute-force leakage model generation attempts to combine potential leakage models based on the fundamental intermediate steps of the algorithm.

These leakage models can be found in a variety of commercial off-the-shelf microcontrollers. A number of microcontrollers have been demonstrated to have side-channel leakage that can be exploited to break AES-ECB mode, with the assumption of having physical access to the device in order to perform the power measurement.

In addition, ChipWhisperer-Lint can be built into an automated leakage detection or testing method. This method can be setup such that new captures are run on physical hardware, validating leakages that might result from changes in compilers or synthesis tool.

References

[1] G. Becker, J. Cooper, E. Demulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, and S. Saab. Test Vector Leakage Assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference*, 2013.

[2] S. Bhasin, J.-L. Danger, S. Guilley, and Z. Najm. NICV: Normalized Inter-Class Variance for Detection of Side-Channel Leakage. In *Hardware and Architectural Support for Security and Privacy*, 2014.

[3] E. Brier, C. Clavier, and F. Olivier. Correlation Power Analysis with a Leakage Model. In *Proceedings of 6th Workshop on Cryptographic Hardware and Embedded Systems (CHES '04)*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg, 2004.

[4] J. Geisler. Apparatus and method for precharging a bus to an intermediate level, Dec. 30 1997. US Patent 5,703,501.

[5] G. Goodwill, B. Jun, J. Jaffe, and R. P. A testing methodologyfor side channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.

[6] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO' 99*, pages 388–397. Springer-Verlag, 1999.

[7] C. Mead and L. Conway. *Introduction to VLSI systems.* Addison-Wesley, 1980.

[8] A. Moradi and T. Schneider. Improved side-channel analysis attacks on xilinx bitstream encryption of 5, 6, and 7 series. In F.-X. Standaert and E. Oswald, editors, *Constructive Side-Channel Analysis and Secure Design*, pages 71–87, Cham, 2016. Springer International Publishing.

[9] C. O'Flynn. *A Framework for Embedded Hardware Security Analysis.* PhD thesis, Dalhousie University, 2017.

[10] C. O'Flynn and Z. Chen. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In *Proceedings of 5th Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE '14)*, volume 8622 of *Lecture Notes in Computer Science*, pages 243–260. Springer International Publishing, 2014.

[11] C. O'Flynn and Z. Chen. Synchronous sampling and clock recovery of internal oscillators for side channel analysis and fault injection. *Journal of Cryptographic Engineering*, 5(1):53–69, 2015.

[12] C. O'Flynn, Z. Chen, and E. Oswald. *Power Analysis Attacks Against IEEE 802.15.4 Nodes*, pages 55–70. Springer International Publishing, Cham, 2016.