Investigating the Effects Various Compilers Have on the Electromagnetic Signature of a Cryptographic Executable

Ibraheem Frieslaar
Council for Scientific and Industrial Research
Pretoria, South Africa
Rhodes University
Grahamstown, South Africa
ifrieslaar@csir.co.za

Barry Irwin
Rhodes University
Grahamstown, South Africa
Council for Scientific and Industrial Research
Pretoria, South Africa
b.irwin@ru.ac.za

ABSTRACT

This research investigates changes in the electromagnetic (EM) signatures of a cryptographic binary executable based on compile-time parameters to the GNU and clang compilers. The source code is compiled and executed on the Raspberry Pi 2 which utilizes the ARMv7 CPU. Various optimization flags are enabled at compile-time and the output of the binary executable's EM signatures are captured at run time. It is demonstrated that GNU and clang compilers produced different EM signature on program execution. The results indicated while utilizing the optimization flag *O3* the EM signature of the program changes. Additionally, the g++ compiler demonstrated fewer instructions were required to run the executable, this related to fewer EM emissions leaked. The EM data from the various compilers under different optimization levels was used as input data for a correlation power analysis attack. The results indicated that partial AES-128 encryption keys was possible. In addition, the fewest subkeys recovered was when the clang compiler was used with level *O2* optimization. Finally, the research was able to recover 15 of 16 AES-128 cryptographic algorithm's subkeys.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Cryptanalysis and other attacks**; **Tamper-proof and tamper-resistant designs**;

KEYWORDS

Electromagnetic, Raspberry Pi, Compilers, CPA, C/C++

ACM Reference format:
Ibraheem Frieslaar and Barry Irwin. 2017. Investigating the Effects Various Compilers Have on the Electromagnetic Signature of a Cryptographic Executable. In *Proceedings of SAICSIT '17, Thaba Nchu, South Africa, September 26–28, 2017,* 10 pages.
https://doi.org/10.1145/3129416.3129436

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAICSIT '17, September 26–28, 2017, Thaba Nchu, South Africa
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5250-5/17/09…$15.00
https://doi.org/10.1145/3129416.3129436

1 INTRODUCTION

It is well known that cryptographic algorithms are used to protect and hide secret information from eavesdroppers. While these algorithms are mathematically secure, the technique of Side Channel Analysis (SCA) to break and recovering secret information from cryptographic algorithms has been well documented [12, 20, 21]. The basic concept of SCA attacks is to measure the power consumption or electromagnetic (EM) emissions of a device while it executes the cryptographic algorithm. The SCA locates a correlation between the consumption and intermediate values of the algorithm which can be used to reveal the secret information [20].

The capturing of EM emissions allows the adversary to mount attacks remotely without the user's knowledge. In addition, EM based attacks are not limited to the traditional attacks against embedded device. However, it can be used against devices running at frequencies ranging in the high MHz or even GHz spectrum [3, 10, 15, 17, 25]. At these higher frequencies of 600MHz – 1GHz, or higher than 1GHz, basic antennas or near-field probes are used to capture the EM signature and uncover secret information.

Today, these high frequency devices are situated in our homes offering a range of functionality from controlling electrical appliances to running the security system of a household. Unknown by the members in the household these devices are being attacked remotely as nations around the world attempt to enforce laws that enables backdoors and remote access to devices or cryptographic schemes. It becomes important to evaluate all aspects of the development chain. One of these aspects is the compilation of source code to executable binaries.

This research investigates the differences in the EM emissions produced from a cryptographic binary that has been compiled by different C/C++ compilers and utilizes various optimization flags. The experiments are performed by utilizing a Raspberry Pi 2 with the ARMv7 CPU. Furthermore, this research aims to answer the following questions:

(1) Do different C/C++ compilers emit different EM emissions?
(2) What is the effects on the EM signature of an executable binary as optimizations are enabled?
(3) Can sensitive information be recovered from these EM emissions?
(4) Would the various compilers optimizations assist in obfuscating information or enhancing the recovery of information?

The remainder of this paper is organized as follows: Section 2 discusses the SCA attacks utilizing EM emissions against high frequency devices; a brief overview of the compilers and the optimization flags is describe in Section 3; followed by Section 4 which details the methodology of this research; the experiments, results and analysis is elaborated in Section 5; and finally the paper is concluded with a discussion in Section 6.

2 ELECTROMAGNETIC ATTACKS

This section discusses the side channel analysis attacks involving electromagnetic (EM) attacks against high frequency devices such as smartphones and systems utilizing ARM processors.

As a program executes on the device, an electrical current passes through that device. These electrical impulses emits EM radiation and subsequently can be used to retrieve secret information [12]. In addition, the adversary does not require direct contact with the device and is able to capture EM data without tampering with the device. Therefore, EM attacks are less intrusive than the conventional attacks via power analysis [3, 14, 15].

A SCA attack utilizing EM data was carried out against a Java based cellphone by Aboulkassimi *et al.* [1]. Unfortunately, they added an intrusive factor by placing a MicroSD extension cable onto the MicroSD card to extract EM information. In addition, Goller and Sigl [17] attacked an Android smartphone executing public key cryptography algorithm. However, the smartphone's shielding plate was removed.

Nakano *et al.* [28] targeted the RSA and elliptic curve cryptography (ECC) encryption implementations contained in the Java Cryptography Extension (JCE) on a smarthpone. The battery and metal covers were removed to assist in recovering EM data. There is no mention of the device specification, only that the device was running at 832 MHz.

Successful attacks against the Elliptic Curve Digital Signature Algorithm (ECDSA) implementation of Android's BouncyCastle library was demonstrated by Belgarric *et al.* [3] and Genkin *et al.* [14] concurrently. However, Belgarric *et al.* followed an intrusive approach by placing the magnetic probe within the smartphone, whereas Genkin *et al.* situated the magnetic probe in close proximity of the device, hence a less intrusive approach. Genkin *et al.* demonstrated the vulnerabilities of an iOS and Android devices where the ECDSA signing keys from OpenSSL were recovered. Furthermore, secret keys from the Corebitcoin application were recovered.

A Differential Power Analysis (DPA) attack against the bitsliced AES encryption algorithm executing on a BeagleBone Black Cortex-A8 processor running at 1 GHz on a development board was successful performed by Balasch *et al.* [2]. Additionally, Longo *et al.* [25] carried out a similar attack against the same development board. Vulnerabilities were demonstrated in both cases against the symmetric key encryption. However, in both studies reveal that the EM probe was physically glued onto the area of leakage and focused on specialized hardware with a proposed hardware countermeasure.

The researchers in [10] demonstrated the ability to recover partial AES-128 cryptographic keys from the Crypto++ library which executed on a Raspberry Pi. The Raspberry was set to operate at a fixed 600 MHz utilizing the Lubuntu 16.04 Operation System. Furthermore, it was demonstrated that the Raspberry Pi was vulnerable against the CPA attack.

This section describes the vulnerabilities high powered devices has to SCA attacks. Additionally the research faced similar issues with the alignment of the captured traces. Table 1 illustrates the alignment techniques used by the fellow researchers. In addition, digital signal processing techniques used to enhance the signal are provided. It is noted that this research aims to investigate the leakage of various compilers and not attempt long range attacks.

The table indicates that the various researchers utilized different techniques in order to align the captured traces. Furthermore, a range of digital signal processing methods were used by the researchers. A fixed technique is not used, thus depending on the potential attack various alignment and digital signal processing techniques can be used.

3 COMPILERS

This section discusses the basic concept of a compiler and elaborates on the inner workings of the GNU Compiler Collection (GCC)[1] and clang[2] compilers.

A compiler translates source code from a high-level programming language to machine code which the Central Processing Unit (CPU) interprets. In addition, compilers forms a link between high-level languages and the underlying hardware. Furthermore, a compiler verifies code syntax, performs run-time operations, and process the output according to the assembler and linker conventions [8].

A compiler consists of three stages [19]. These stages are the front, middle, and back end. The syntax and semantics are verified in the front end. In addition, lexical and syntax analysis are performed to inform the user of any errors or warnings in the source code. Furthermore, the front end generates an intermediate representation (IR) of the source code which is processed by the middle end.

Optimizations are carried out in the middle end such as the removal of useless code, repositioning of loops that require less resources, and so forth. This results in an optimized IR which is sent to the back end.

The optimized IR is further analyzed and additional optimizations are performed in the back end. These may include optimizing the code for a particular target hardware and the out of this stage produces machine code specialized for a particular processor and operating system.

3.1 GNU Compiler Collection

The GCC compiler is produced by the GNU Project. The standard compiler for most Unix Operating Systems is the GCC compiler [34]. Two option are available to compile C/C++ code – gcc for C and g++ for C++ source code – The gcc and g++ commands invokes the compiler which converts the source code into machine code and produces a complete executable binary.

These compilers utilizes the three stage process. A parser is used to produced syntax trees from the source code in the front end.

[1] https://gcc.gnu.org/
[2] https://clang.llvm.org/

Table 1: Various trace alignment and digital signal processing techniques.

Research	Alignment Techniques	Signal Processing
Aboulkassimi *et al.* [1]	Spectral Density based Approach (SDA) and Template based Resynchronization Approach (TRA)	
Nakano *et al.* [28]		High & Low pass filters, Kalman filtering, and Threshold-based Wavelets transforms.
Balasch *et al.* [2]	Acknowledge traces were misaligned. However, there was no mention of alignment techniques.	
Longo *et al.* [25]	Least squares technique.	
Belgarric *et al.* [3]	A sleep function was used.	FIR (Finite Impulse Response) filter weighted with a Hamming window at a cutoff frequency of 50kHz. Followed by a high-pass filter.
Genkin *et al.* [14]	Locate specific patterns and create a template which locates reassurances of the specific pattern in other traces.	Singular Spectrum Analysis (SSA). Followed by an FIR low–pass filter to suppress noise outside the 0–125 kHz band.
Frieslaar and Irwin [10]	Elastic alignment, peak detection and the sum of absolute differences (SAD). Savitzky and Golay filters.	Low and high pass filters was utilized. Followed by, quadratic demodulation

GCC first utilized LALR parsers generated with Bison [23]. However, hand-written recursive-descent parsers [27] are used today to support C++ source code. The trees are converted to the middle end's input representation of language-independent trees which is known as GENERIC or GIMPLE form [29] . GENERIC is more complex as apposed to GIMPLE , which is a simplified GENERIC, where various constructs are lowered to multiple GIMPLE instructions.

Compiler optimizations and static code analysis techniques are applied to the trees. These work on multiple representations, mostly the architecture-independent GIMPLE representation and the architecture dependent RTL representation. Finally, machine code is produced using architecture-specific pattern matching .

3.2 Clang

Clang was designed to replace the GCC compiler [22]. It works in conjunction with the Low Level Virtual Machine (LLVM) [22]. This combination allows to replace the full stack of the GCC compiler. It is based on a library-based design which facilitates in the incorporation of other applications.

Since clang is a library-based architecture it aids the compiler to be integrated with tools that interact with source code, such as an integrated development environment (IDE) graphical user interface (GUI). In comparison with GCC, which utilizes compile-link-debug cycle, it becomes difficult to integrate with IDEs.

More information is retained during the compiling process as apposed to GCC. The original code is preserved, which make it easier to map errors back into the original source. The error reports are more detailed and specific in such a way that IDEs can index the output of the compiler during compiling. Furthermore, the parse tree supports automated code refactoring, as it directly represents the original source code.

3.3 Compiler Optimizations

As mentioned before the compilers has various optimization [13, 24] which can be used to increase the execution time. Table 2 displays the optimizations for the the GNU and clang compilers. To enable an optimization the *-O* is used followed by the optimization flag [32]. As seen in the table, the optimization flags can range from *O1 – O3*, *Os*, *Ofast*, *Og* and *Oz*. Each flag has it's own optimization parameters. An example of invoking the parameters would be "*gcc -O3*" or "*clang -O1*".

Table 2: A comparison between the optimizations of the GNU and clang compilers.

Flag	Compiler	
	gcc/g++	**clang**
0	No optimization is performed and the source code is complied in the most straightforward way possible.	No optimization is performed at this level, the compiler generates the fastest and most debuggable code.
1	This level enables the most common forms of optimization which does not require any speed–space tradeoffs. With this option the resulting executables should be smaller and faster than with *-O0*.	A minimal optimization level which lays between *O0* and *O2*.
2	This option turns on further optimizations, in addition to those used by *O1*. These additional optimizations include instruction scheduling. Only optimizations that do not require any speed-space tradeoffs are used.	This is a moderate level of optimization which enables most optimizations and reduces the code size.
3	This option turns on more expensive optimizations, such as function inlining. The speed of the resulting executable is increased. However, the size is increased as well. Under some circumstances where these optimizations are not favorable, this option might actually make a program slower.	This is built on the previous optimization, except that it enables optimizations that take longer to perform or that may generate larger code.
s	*Os* enables all *O2* optimizations and performs further optimizations reducing source code size.	The same as *O2* with extra optimizations to reduce code size
fast	*-Ofast* enables all *O3* optimizations. Furthermore, it enables optimizations that are Fortran-specific.	
g	Enables optimizations for debugging	The same as *O1* with debugging optimizations
z		*Oz* is based on *Os* but reduces code size further.

Optimizations are mainly used to increase the performance of a program. However, this research aims to investigate the possibility of utilizing the optimizations as a security feature in terms of protecting against SCA attacks. Furthermore, this research will focus on the optimizations ranging from *O1 – O3*.

4 METHODOLOGY

This section details the methods used to acquire, process and utilize data in order to recover secret information from a Raspberry Pi. The Raspberry Pi has been selected as it can used for a range of functionality such as home automation [18], industrial usage [31] and sensors used to retrieve critical information [7], amongst others. Since the device can be utilized in a variety of tasks, it is imperative to evaluate this device as any comprise to the device can lead to a major data breach. Furthermore, the equipment used in this research is discussed in Section 4.1.

Figure 1 illustrates the process followed to retrieve the secret information. The procedure consists of three phase: the data acquisition; post-processing and attack phase. These phases will be further discussed in Sections 4.2 – 4.5.

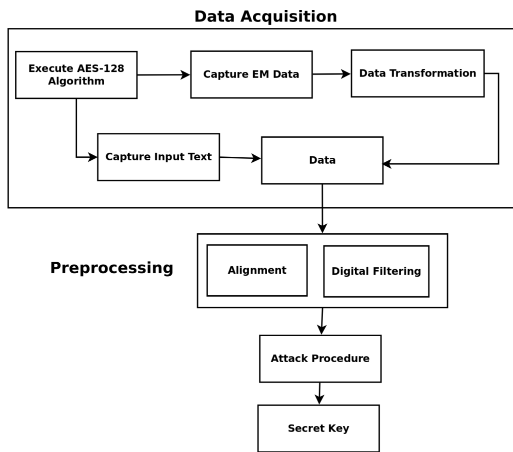
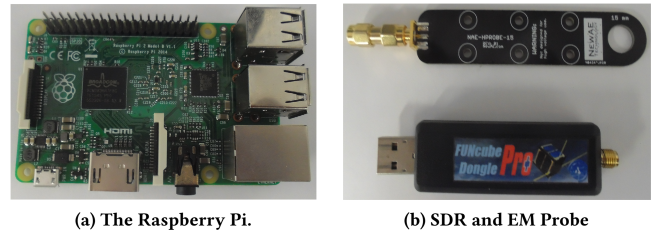


Figure 1: The flow diagram of the attack procedure.

4.1 Equipment

The research makes use of the Raspberry Pi 2 Model B which is the second generation Raspberry Pi [30]. It includes A 900MHz quad-core ARM Cortex-A7 CPU with 1 GB RAM. The ARMv7 processor, enables the device to install a full range of ARM GNU/Linux distributions. Furthermore, the ARM A7 architecture is found in many smartphones.

Two Raspberry Pi 2's, the FUNcube Dongle Pro+ [11] software defined radio (SDR) and a CW505 Planar H-Field Probe [5] was used in this research to recover EM data from a Raspberry Pi. The FUNcube dongle Pro+ and the H-Field probe is depicted in Figure 2b. The FUNcube dongle Pro+ dongle receiver contains a software-adjustable mixer and a 192kHz ADC, accessed via USB as a sound-card audio interface to receive radio frequencies. It has a range of presents for the input bandwidth, ranging from 44.1 kHz – 384 kHz.



(a) The Raspberry Pi. (b) SDR and EM Probe

Figure 2: Additional equipment used. (a) The Raspberry Pi 2 Model B and (b) The FUNcube SDR dongle Pro+ (r) with the CW505 Planar H-Field Probe (l)

For more information on the dongle, the reader is referred to the specifications page [11].

4.2 Data Acquisition

This section discuses the approach this research used to retrieve useful information from the Raspberry Pi. The procedure can be seen in Figure 3. This process is part of the capture EM data and data transformation as seen in Figure 1.

The procedure starts by capturing raw signals using the SDR. The raw signal is processed in GNURadio [16] where a low and high pass filter is applied. GNURadio is a free software development toolkit that provides signal processing blocks to implement software-defined radios and assist in signal-processing systems. The resultant signal is sent to a signal processing program called Baudline where the region of interest is extracted. The region of interest is sent back to GNURadio where quadratic demodulation is applied, followed by another low pass filter. Upon completion the data is saved and transfered to the post processing phase as depicted in Figure 1.

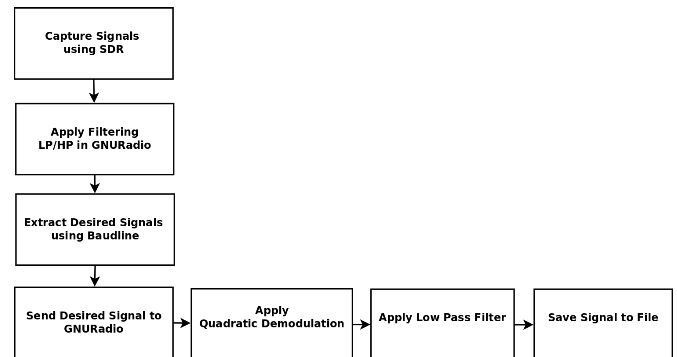


Figure 3: The flow diagram of the process used to extract useful information from the Raspberry Pi.

The EM data capturing comprised of utilizing two Raspberry Pi's. The first device served as the victim, while the secondary Pi was the attacker. The Lubuntu 14.04 operating system with the Linux 3.18.0-20-rpi2 kernel were used on both devices. No services in the Operating System was disabled. Furthermore, to limit the CPU from using internal step-up controls to adjust power and CPU

frequency the victim's maximum CPU frequency was configured to 600 MHz. This research followed the approach by [14] and [2] to limit the CPU frequency. Furthermore, no adjustments were made to the secondary device.

The FUNcube dongle was inserted into a USB port in the attack device and GNURadio was used to interface with the device. Figure 4 illustrates the experimental setup, on the right is the attacker with the FUNCube dongle and the H-Field probe connected to it. While the EM probe is placed over the CPU of the victim.

Figure 4: The experimental setup of the two Raspberry Pi's

4.3 Analyzing Data

This section elaborates on the techniques used to analyze the captured EM emissions into meaningful information.

There are three stages that the EM data can be analyzed [9, 28]. The first stage consists of computing the fast Fourier transform (FFT) over the baseband waveform. This process establishes a frequency signature for various operations, as different operations produces a specific pattern. Figure. 5 illustrates the raw signal after it has been processed through a FFT. The signal was obtained by monitoring the desired 600 MHz frequency in real time via the SDR.

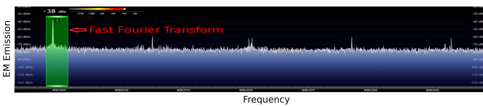


Figure 5: The fast Fourier transform at 600 MHz.

The second stage is to determine the region of interest at the point of leakage, which can be visually determined. Figure 6 depicts the amplitude domain, with the arrow pointing to the power spike which is the point of interest. A more defined representation of the point of interest is demonstrated in Figure 7.

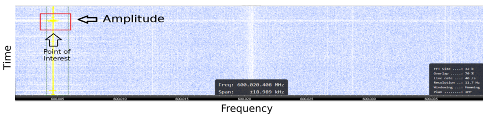


Figure 6: The point of interest in the amplitude domain at 600 MHz.

Figure 7 depicts the data in the amplitude domain, rotated 90°for a better visual representation. The region of interest is displayed in the figure by a rectangle. This illustrates to adversary the location in time and the type of EM signature produced for a certain procedure.

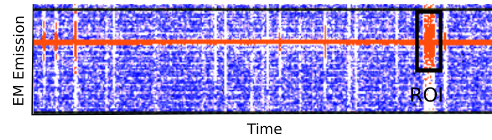
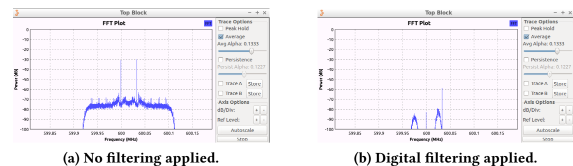


Figure 7: The region of interest highlighted in the amplitude domain as an execution is performed on the Raspberry Pi.

The third stage is to apply digital filtering which removes noise in the signal. Once the noise has been removed, the signal can be evaluated. This will be further discussed in detail in Section 4.4.

While the victim executes the cryptographic program, the adversary uses GNURadio to intercept the EM emissions from the Raspberry Pi. The adversary captures the EM emissions per execution. The data is captured at 384kHz, as sampling at a higher rate allows for more data to be obtained. A low and high pass filter with a cut off frequency of 25kHz and a transition width of 15kHz was applied to the signal. Once the execution of the test programs completes, the signal data is stored and sent to Baudline.

The point of interest is extracted and sent back to GNUradio for further digital processing where quadratic demodulating is applied which is followed by, an additional low pass filter with 37.5kHz as a cut off frequency and 18.75kHz as transition. The cut off frequency of 37.5kHz is determined by utilizing the Raspberry Pi's baudrate and dividing it by three and the transition width is determined by dividing the cutoff frequency of 37.5kHz by two. Figure 8a demonstrates the signal passed through the FFT with no filter, followed by Figure 8b which illustrates the signal after it has been through a low and high pass filter.



(a) No filtering applied. (b) Digital filtering applied.

Figure 8: A comparison of the signal (a) no filtering and (b) with filtering

4.4 Preprocessing

This section describes the techniques used to transform the usable data into data that can be used by the attack procedure to recover the secret key from a AES-128 cryptographic implementation on a Raspberry Pi. These procedures form part of the prepocessing procedure depicted in Figure 1.

The data acquired in the previous section was captured unsynchronously thus, the data needs to be aligned. Figure 9 depicts the process used to align the data, as well as recovering secret information. Furthermore, this research follows the approach by [10] to align the captured EM data from a Raspberry Pi.

Firstly, the signal is segmented into three smaller partitions. Each partition is aligned by using the elastic alignment technique [33]. After alignment the signal is concatenated into a one large signal.

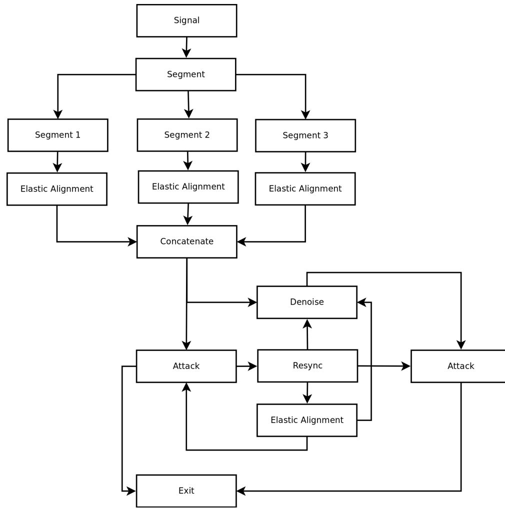


Figure 9: The flow diagram to align the signal and recover secret information [10].

The resultant signal is used as input data for the attack or it can be sent to the denoising procedure. If the attack procedure is successful, i.e: secret information has been recovered, exit the routine, or else move to the resync procedure. Peak detection and the sum of absolute differences (SAD) techniques are used to compensate for trigger jitter and phase shift in order to resync the data. Once the resyncing procedure is completed the adversary could either send the data to attack procedure or apply denoising to the data. Furthermore, he could apply elastic alignment before sending the data to the attack procedure.

4.5 Attack Procedure

This section discusses methods used in the attack procedure as first depicted in Figure 1 to retrieve the secret keys of the cryptographic implementation of AES-128 algorithm. The Correlation Power Analysis (CPA) attack is used as the side channel analysis attack.

The correlation equation is required to determine a correlation between the key guesses and the EM emissions. For a detailed explanation of the mathematical approach of the correlation equation the reader is referred to [4]. The correlation equation is as follows.

$$r_{i,j} = \frac{\sum_{d=1}^{D}[(h_{d,i} - \bar{h_i})(t_{d,j} - \bar{t_j})]}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i})^2 \sum_{d=1}^{D}(t_{d,j} - \bar{t_j})^2}} \quad (1)$$

Where $t_{d,j}$ is the captured EM trace, with the total number of traces D and $h_{d,i}$ is the hypothetical values produced by the Hamming weight power model [26].

The attack on the AES-128 cryptographic algorithm commences as the secret key is sent to the S-Box round of the AES-128 algorithm. The Hamming weight power model is implemented with a guessing procedure. The system iterates one subkey at a time and guesses every possible outcome for that subkey. These guess values range from 0 – 255. The next phase is to calculate the corresponding intermediate value of that guess. The value of each guess is converted to its binary representation of the value, with the total number of 1's summed up to determine the *weight*. This is referred to the hamming weight power model [26]. The AES-128 algorithm has 16 subkeys [6] and since each subkey is attacked one at a time there is only 2^{12} possibilities, instead of 2^{128} possibilities to predict the correct secret key.

It is noted that a negative correlation is possible. However, the absolute value is taken. Furthermore, to achieve a ranking system the correlation of each guess is stored and the guess with the maximum correlation is predicted to be the subkey.

5 EXPERIMENTS, RESULTS, AND ANALYSIS

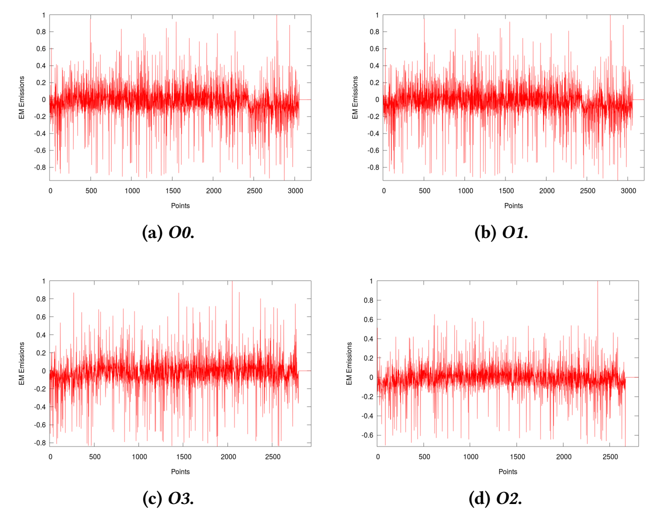
This section discusses the impact the various C/C++ compilers had on the EM spectrum. The compilers used was the gcc, g++, and clang compilers. Furthermore, this section comprises of Section 5.1 where the EM signatures from the various compilers are discussed and Section 5.2 details the recovery of secret information by utilizing the EM emissions from a Raspberry Pi.

5.1 EM Leakage

This section discusses the experiments and results obtained from capturing the EM data from the AES-128 cryptographic programs. Each program was compiled by the various compilers

The experiments consisted of compiling the AES-128 algorithm of libcrypto++ within the *pThread* environment [10]. In addition, the compilers were set to use various optimization parameters. Furthermore, the EM emissions were captured as the program executed.

The first set of experiments focused on compiling the program with g++ with the optimizations flag *O*. The g++ optimizations range from *O0* – *O3*. Figure 10 depicts the EM leakage from the cryptographic program compiled with different optimization flags.

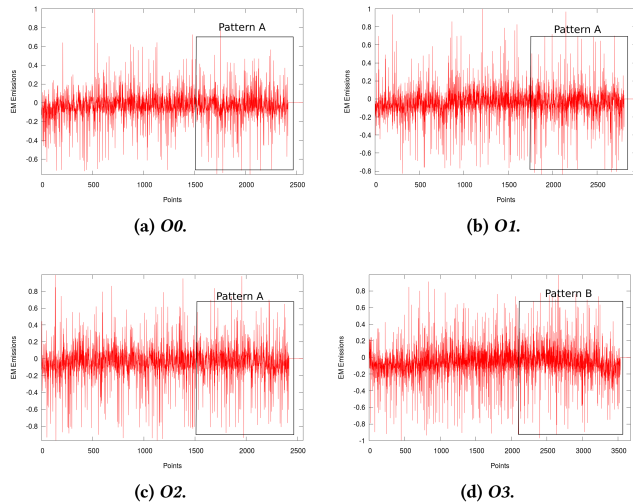


(a) *O0*. (b) *O1*. (c) *O3*. (d) *O2*.

Figure 10: The EM leakage from the cryptographic program compiled with g++ using different optimization flags.

It is observed a similar pattern occurs from Figures 10a – 10c, these are the optimizations flags of *O0*, *O1*, and *O3*. However, the optimization flag of *O2* resulted in a different EM signature than the rest which is evident Figure 10d where the power spikes are less profound.

The next set of experiments consisted of investigating the effects the gcc compiler had on the EM signatures. The gcc optimizations range from *O0* – *O3*. Figure 11 illustrates the EM leakage from the cryptographic program compiled with gcc using different optimization flags.



(a) *O0*. (b) *O1*. (c) *O2*. (d) *O3*.

Figure 11: The EM leakage from the cryptographic program compiled with gcc using different optimization flags.

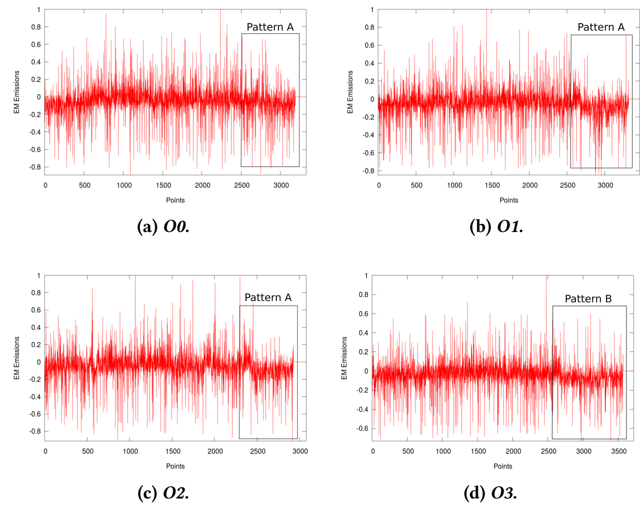
Figures 11a – 11c depicts that using the optimization flags of *O0* – *O2*, has a similar EM signature. However, Figure 11d – *O3* flag – demonstrates a different EM signature from the rest. More data points are used, thus relating to a greater power draw as more spikes are prevalent as well.

The third set of experiments focused on the effects the clang compiler had on the EM signatures. The clang optimizations range from *O0* – *O3*, *Os*, *Ofast*, and *Oz*. The EM leakage from the cryptographic program compiled with clang using different optimization flags is depicted in Figure 12.

Figures 12a – 12c illustrates utilizing the optimization flags of *O0* – *O2*, has a similar EM signature. However, Figure 12d – *O3* flag – demonstrates a different EM signature from the rest. A greater power draw was used as seen by the increase in data points, i.e: the data points end at 3500.

Figure 13 starts by illustrates the point of spawning the *pThread* inside the main method for the g++ output.

Figure 13a displays that utilizing the optimization *O0* resulted in the use of the *str*, *load*, and *cmp* instructions as apposed to only using a *mov* instruction as seen in Figures 13a and 13b marked by the *A* pointing to the code, respectively. Before reaching the instruction set of *(vcvt.f32.s32 s14, s13)*, the optimization *O3* only had two instructions where as with the flag *O0* enabled, many instructions were required. This relates back to the data in Figure



(a) *O0*. (b) *O1*. (c) *O2*. (d) *O3*.

Figure 12: The EM leakage from the cryptographic program compiled with clang using different optimization flags.

10d as fewer data points were used and the power spikes was fewer as the optimization flag *O3* was used. This is further confirmed from Table 2 as optimization *O3* reduces execution time.

The comparison between the assembly code using optimization flags *O0* and *O3* for the clang compiler is displayed in Figure 14.

A reoccurring theme is illustrated as the code indicates that utilizing the optimization flag *O3* produces less machine code. Although, the code is shorter, there are additional instructions such as *(vcvt.f64.f32)* used in optimization *O3*, depicted in Figure 14b by the annotation *A* pointing to the instruction. The *(vcvt.f64.f32)* instruction converts between single-precision and double-precision numbers. This explains why the EM signature in Figure 12d was different from the rest of the EM signatures produced by the clang compiler as double precision uses twice as many bits as single, 32 bits for a single and 64 bits for a double. As more bits are used, more bus lines are required, hence more power(data points) are generated.

This section discussed the effects the gcc, g++, and clang compilers had on the EM signature of a cryptographic program. The AES-128 cryptographic program within the multi-threaded framework of *pThreads* was used as the base program. As the program was compiled, different optimization flags were used. The results indicated that using the optimization flag of *O3* changes the EM signature of the program. Furthermore, the assembler code was analyzed and the g++ implementation displayed that fewer instructions were required which related to fewer EM emissions leaked. Although, fewer instructions were used by the clang compiler the EM trace had more data points which ties back to Table 2 as it is possible that the code would have a longer run time. Furthermore, The analysis showed that using the optimization flag of *O3* in the clang compiled had enable the usage of double precision numbers which required more bits and power to be used.

(a) Optimization flag *O0*.

(b) Optimization flag *O3*

Figure 13: Comparison between the assembly code using optimization flags (a) *O0* and (b) *O3* for the gcc compiler.

(a) Optimization flag *O0*.

(b) Optimization flag *O3*

Figure 14: Comparison between the assembly code using optimization flags (a) *O0* and (b) *O3* for the clang compiler.

5.2 Recovering Secret Information

The section discusses the experiments and results performed to retrieve secret information from a Raspberry Pi while executing the AES-128 cryptographic program. Furthermore, the programs was compiled with gcc, g++, and clang while utilizing the various optimization parameters.

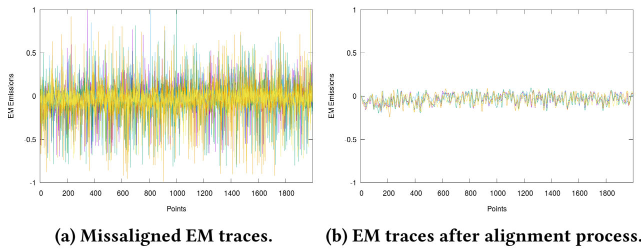
The experiments consisted of capturing the EM emissions from the Raspberry Pi while the cryptographic executable ran. The programs were repeated 30 occasions, hence 30 EM traces were captured. The same procedure mentioned in Section 5.1 was utilized to collect and process the data.

Once the data was collected the preprocessing procedure in Figure 1 was applied. The preprocessing procedure as mentioned in Section 4.4 applies various techniques such as elastic alignment, denoising, and resyncing techniques to align the data.

The in depth process to align the data can be seen in Figure 9. Furthermore, Figure 15a illustrates the EM traces before the alignment process and Figure 15b after the alignment process has been applied.

The resultant data is sent to the attack procedure as discussed in Section 4.5. The data was used as input for the CPA attack. The first experiment involved utilizing 10 traces as input. Table 3 depicts the results achieved as 10 traces were used as input.

The table comprises of four columns and rows. The first column represent the optimization flag used by the compiler, followed by columns two – four, depicting the number of subkeys recovered



(a) Missaligned EM traces. (b) EM traces after alignment process.

Figure 15: Comparison between the (a) the missaligned EM traces and (b) the EM traces after the alignment process has been applied.

from the EM data. It is known that there are a total of 16 subkeys for the AES-128 cryptographic algorithm.

Table 3: The subkeys recovered utilizing 10 traces.

O. Flags	g++	gcc	clang
0	7	7	5
1	5	7	3
2	3	3	1
3	2	2	5

Table 3 reveals an interesting pattern, as more optimization were used the key recovery decreased as seven subkeys was recovered while no optimization were used and two subkeys was recovered as level three optimization were used for the gcc/g++ compilers. Furthermore, the clang compiler led to decrease of the number of subkeys recovered as more optimizations were used. However, *03* optimization demonstrated that the same number of subkeys was retrieved as no optimizations was used. This relates back to Table 2 where level three optimization increases the execution time of a program and further evident in Figure 12d where more data points are produced in the EM signature. In addition, it is shown that the clang compiler with level two optimization reveal the least secret information.

Table 4 and 5 depicts the results of the CPA attack as additional EM data was added to input data for the attack.

Table 4: The subkeys recovered utilizing 20 traces.

O. Flags	g++	gcc	clang
0	9	10	6
1	7	9	5
2	6	5	2
3	4	5	7

Table 5: The subkeys recovered utilizing 30 traces.

O. Flags	g++	gcc	clang
0	12	12	9
1	10	11	7
2	8	8	5
3	7	7	9

The results in both tables reveals a reoccurring theme. The more input data used for the CPA attack, the greater the number of subkeys recovered. Furthermore, as the optimization levels increase less subkeys are recovered as apposed to using no optimizations flags. Additionally, the clang compiler reveals the fewest subkeys.

The subkeys that were recovered in the previous experiments were further analyzed. Table 6 and 7 depicts the recovered subkeys for the GNU and clang compiler under the different compiler optimizations.

Table 6: The subkeys recovered from the G++ compiler over the various optimizations while utilizing 30 EM traces.

	Subkey															
Flag	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
0	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	Y	Y	Y
1	Y	Y	Y	-	Y	Y	-	Y	-	Y	Y	-	-	Y	Y	Y
2	Y	Y	-	-	-	-	Y	Y	-	-	-	Y	-	Y	Y	Y
3	Y	Y	Y	Y	-	-	Y	Y	-	-	-	-	-	Y	Y	-

Table 7: The subkeys recovered from the clang compiler over the various optimizations utilizing 30 traces.

	Subkey															
Flag	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
0	Y	Y	Y	Y	Y	-	Y	-	-	Y	-	-	-	-	Y	Y
1	Y	-	-	-	-	-	-	Y	-	Y	Y	-	-	Y	Y	Y
2	Y	-	-	-	-	-	Y	Y	-	-	-	-	-	Y	Y	Y
3	-	Y	Y	Y	Y	-	Y	Y	-	-	-	-	-	Y	Y	Y

Table 6 depicts that utilizing no optimizations for the GNU compiler subkeys 1 – 9, and 14 – 17 were recovered. Although, fewer subkeys were recovered while utilizing optimizations *O1* and *O2*, subkeys 10–12 were retrieved. A similar observation is made for the results of the clang compiler as seen in Table 7. Fewer subkeys are recovered. However, different subkeys in the secret key were being retrieved.

This section has demonstrated the ability to recover partial AES-128 cryptographic keys from a Raspberry Pi. In addition, the cryptographic program was compiled with various C/C++ compilers and optimization levels. The CPA attack revealed that the GNU compilers achieved a similar result with respect to the number of subkeys recovered. However, it was demonstrated that the subkey location changed as different optimizations were used. Summing up, the recovered subkeys, the researchers were able to retrieve 15 of the 16 AES-128 subkeys, with only subkey 13 not being recoverable. Furthermore, the fewest number of subkeys was recovered as the clang compiler was used with *O2* optimization. Although, utilizing higher levels of optimization, the results revealed that *O3* optimization for clang had the same effect as using no optimizations.

6 CONCLUSION

This research has successfully answered and provided an in depth analysis to the questions posed in Section 1. The first two questions being "Do different C/C++ compilers emit different EM emissions?" and "What is the effects on the EM signature of an executable binary as optimizations are enabled?". The research demonstrated that GNU and clang compilers produced different EM signature as a cryptographic program executed by demonstrating that the

various optimization flags changes the EM signature of the program. The assembler code was analyzed and the g++ implementation displayed that fewer instructions were required which related to less EM leakage. In addition, the clang machine code depicted that different instruction sets were used and related to more EM data being radiated.

The next set of question posed were "Can sensitive information be recovered from these EM emissions?" and "Would the various compilers optimizations assist in obfuscating information or enhancing the recovery of information?". Therefore, the EM data from the various compilers under different optimization levels was used as input in a CPA attack and partial AES-128 encryption keys was recovered. The CPA attack revealed that the GNU compilers achieved a similar result with respect to the number of subkeys recovered. However, the location of the subkey recovered had varied. Summing up, the recovered subkeys, the research was able to retrieve 15 of the 16 AES-128 subkeys. Additionally, the fewest subkeys was recovered while utilizing the clang compiler with *O2* optimization. Although, utilizing more optimization, the results revealed that level *O3* optimization for clang had the same effect as utilizing no optimizations.

This research clearly demonstrates that the various compiler optimizations affects the EM signature of the binary executable. This is important as this information can be utilized by developers and system engineers to defend against known and future SCA attacks by optimizing compiler parameters in a configuration that can emit EM emissions to obfuscate adversaries. In addition, this research has recovered three additional keys, with a total of 15 subkeys as apposed to the work in [10] were only 12 subkeys were recovered.

FUTURE WORK

Although, various optimization levels are enable by the *O0 – O3* flags. There is still room to investigate the effect of individual optimization flags such as *-fPIE*, *-D_FORTIFY_SOURCE=2*, *-fstack-protector*, and more which may provide protection to the binary, which is largely linked to memory protection

ACKNOWLEDGEMENT

This work was undertaken as part of the Distributed Multimedia CoE at Rhodes University, with financial support from the Information Security Competency Area within Modelling and Digital Science at the CSIR, Telkom SA, Tellabs/ CORIANT, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 90243). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

REFERENCES

[1] Driss Aboulkassimi, Michel Agoyan, Laurent Freund, Jacques Fournier, Bruno Robisson, and Assia Tria. 2011. ElectroMagnetic analysis (EMA) of software AES on Java mobile phones. In *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*. IEEE, 1–6.

[2] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. 2015. DPA, bitslicing and masking at 1 GHz. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 599–619.

[3] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. 2016. Side-channel analysis of Weierstrass and Koblitz curve ECDSA on Android smartphones. In *Cryptographers Track at the RSA Conference*. Springer, 236–252.

[4] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 16–29.

[5] ChipWhisperer. 2016. CW505 Planar H-Field Probe. (2016). https://wiki.newae.com/CW505_Planar_H-Field_Probe

[6] Joan Daemen and Vincent Rijmen. 2005. Rijndael/AES. In *Encyclopedia of Cryptography and Security*. Springer, 520–524.

[7] Sheikh Ferdoush and Xinrong Li. 2014. Wireless sensor network system design using Raspberry Pi and Arduino for environmental monitoring applications. *Procedia Computer Science* 34 (2014), 103–110.

[8] Christopher W Fraser and David R Hanson. 1995. *A retargetable C compiler: design and implementation*. Addison-Wesley Longman Publishing Co., Inc.

[9] Ibraheem Frieslaar and Barry Irwin. in press, August, 2017. Investigating the Electromagnetic Leakage from a Raspberry Pi. In *Information Security South Africa (ISSA)*.

[10] Ibraheem Frieslaar and Barry Irwin. in press, September, 2017. Recovering AES-128 Encryption Keys from a Raspberry Pi. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*.

[11] FUNcube. 2013. Specifications. (2013). http://www.funcubedongle.com/

[12] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic analysis: Concrete results. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 251–261.

[13] GCC. 2013. GCC optimization. (2013). https://wiki.gentoo.org/wiki/GCC_optimization

[14] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1626–1638.

[15] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. *Journal of Cryptographic Engineering* 5, 2 (2015), 95–112.

[16] GNURadio. 2013. GNURadio. (2013). https://www.gnuradio.org/

[17] Gabriel Goller and Georg Sigl. 2015. Side channel attacks on smartphones and embedded devices using standard radio equipment. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 255–270.

[18] Sarthak Jain, Anant Vaibhav, and Lovely Goyal. 2014. Raspberry Pi based interactive home automation system through E-mail. In *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*. IEEE, 277–280.

[19] Stephen C Johnson. 1979. A tour through the portable C compiler. *Unix programmer's manual*, 2 (1979).

[20] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Advances in Cryptology–CRYPTO'99*. Springer, 388–397.

[21] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *Journal of Cryptographic Engineering* 1, 1 (2011), 5–27.

[22] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

[23] John Levine. 2009. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.".

[24] LLVM. 2013. LLVM's Analysis and Transform Passes. (2013). http://llvm.org/docs/Passes.html

[25] J. Longo, E. De Mulder, D. Page, and M. Tunstall. 2015. SoC it to EM: electromagnetic side-channel attacks on a complex system-on-chip. Cryptology ePrint Archive, Report 2015/561. (2015).

[26] Hassen Mestiri, Noura Benhadjyoussef, Mohsen Machhout, and Rached Tourki. 2013. A Comparative study of power consumption models for CPA attack. *International Journal of Computer Network and Information Security* 5, 3 (2013), 25.

[27] Hanspeter Mössenböck. 1991. A generator for production quality compilers. *Compiler Compilers* (1991), 42–55.

[28] Yuto Nakano, Youssef Souissi, Robert Nguyen, Laurent Sauvage, Jean-Luc Danger, Sylvain Guilley, Shinsaku Kiyomoto, and Yutaka Miyake. 2014. A pre-processing composition for secret key recovery on android smartphone. In *IFIP International Workshop on Information Security Theory and Practice*. Springer, 76–91.

[29] Diego Novillo. 2006. GCC–An Architectural Overview, Current Status, and Future Directions. In *Proceedings of the Linux Symposium*, Vol. 2.

[30] Raspberry Pi. 2015. Raspberry Pi 2 model b. *Online. https://www. raspberrypi. org/products/raspberry-pi-2-model-b/* (2015).

[31] Paulo Simões, Tiago Cruz, Jorge Gomes, and Edmundo Monteiro. 2013. On the use of Honeypots for detecting cyber attacks on industrial control networks. In *Proc. 12th Eur. Conf. Inform. Warfare Secur. ECIW 2013*.

[32] Richard M Stallman et al. 1989. Using and porting the GNU compiler collection. *Free Software Foundation* 51 (1989), 02110–1301.

[33] Jasper GJ van Woudenberg, Marc F Witteman, and Bram Bakker. 2011. Improving differential power analysis by elastic alignment. In *Cryptographers' Track at the RSA Conference*. Springer, 104–119.

[34] William Von Hagen. 2011. *The definitive guide to GCC*. Apress.