DEPARTMENT OF COMPUTER SCIENCE

Setting the Foundation for Side Channel Attacks on Blackberry Handheld Devices

Omar Saleh

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

---

Autumn 2009 | MS-53

Acknowledgment:

I extend my gratitude to Dr. Elisabeth Oswald and Dr. Michael Tunstall for their guidance and assistance in different aspects of this project. Without their patience, proactive support, welcoming attitude and sheer experience, none of this progress would have been made to reach such a state in tackling a device virtually wrapped in obscurity.

I also wish to thank the cryptography team in the University of Bristol for their warm welcome and support and for making both theoretical and practical aspects of the field thoroughly enjoyable to all.

Special thanks go to my family for their love, support and guidance.

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

_____

Omar Saleh, October 2009

Abstract

Blackberry (BB) devices control the smart phone market for corporate and government agencies, arguably due to the controllable level of protection provided for data stored on the mobile device and for that traveling over the air (OTA). As such, BB cryptographic functionality would be a key element for its position in the market place. Side channel attacks (SCA) have the potential power to extract cryptographic keys from down to one or few traces. Thus, if exploited this could have a devastating effect in undermining the entire BB security model. However, there is a significant gap before launching such powerful attacks can become a reality, since no previous work has been done to set the environment needed to do so. As such, this study aims to facilitate for conducting extensive SCA research on BB, by providing the integration tools needed, locating and isolating electromagnetic emanations (EM) leaked during cryptographic activity, ruling out some possible implementation and providing optimized and automated methods for signal alignment and identification of the Advanced Encryption Standard (AES) that is extensively used in BB's security model.

Table of Contents

Aims and Objectives:

This project aspires to provide an excellent starting point to plan and mount various types of side channel attacks (SCA) on Blackberry handheld devices. This aim is hoped to be realized by accomplishing the following objectives:

1- Preliminary analysis of BB security, and the schemes set to accomplish this goal.
2- Preliminary analysis for BB development environment and providing some guidelines for BB SCA Malware Design.
3- Physical preparation of the mobile device, including finding and isolating electromagnetic emanations (EM) adequate to identify those relating to cryptographic activity.
4- Proposing a flexible model for automated signal acquisition, and integration to current open source SCA software developed at the University of Bristol: Open SCA.
5- Providing an introduction to SCA and how some of them work, aiming to guide planning for a number of potentially effective attacks on a BB.
6- Visual inspection of EMs generated during the execution of the Advanced Encryption Standard (AES) on a BB device for feature extraction and to rule out possible implementations and mitigation techniques, and also identify potential anomalies in the signal. Here, the AES routine's code is proprietary to BB, and implementation details are hidden.
7- Automated and optimized signal alignment on the first round with use of special features found in the signal.
8- Distinguishing AES signals in a live feed with limited training data, and with high performance

Chapter 1: Side Channel Attacks

1.1- Introduction

Traditional attacks on ciphers, including linear, differential and algebraic cryptanalysis techniques, by which only the starting point (plaintext) and ending point (cipher text) can be observed, must deal with the complexity of attacking the cipher as a whole, thus limiting their effectiveness or requiring a great deal of computational resources in order to extract the encryption keys used; especially considering how many modern day ciphers have been designed to be resistant and are sometimes effectively immune against some of those attacks.

Side channel attacks on the other hand are performed by examining encryption as it happens, during which the byproducts, inner workings and physical properties of cryptographic devices are exploited to reveal secret key information. This provides a great deal of power to the attacker, as ciphers -considered strong as a whole- can be analyzed in a divide and conquer fashion, and intermediate stages can be isolated and targeted separately to extract secret key information.

1.2- Types of SCA:

In this section, we shall provide an overview and a brief description of a number of non-invasive SCAs that could represent a potential threat to BB devices.

1.2.1- Power Consumption Attacks:

Complementary Metal Oxide Semiconductor (CMOS) technology is the dominant technology used in building virtually all digital circuits[1], and is the one used in the ARM7 processor. CMOS cells consume both constant “static power” to maintain a given bit state 0 or 1, and noticeably higher “dynamic power” to flip bit states from 0->1 or 1->0[1].

Power attacks are more likely to be feasible when the dynamic power of the entire integrated circuit (IC) dominates the total measurable IC power consumption, which also includes and could be otherwise dominated by glitches and other physical phenomena associated with the assembly and design of the IC. This is assumed as attackers have limited resources in having access to an accurate power consumption model which compensates for those effects as well.

Measuring power consumption will result in getting a high frequency analogue signal, which can be sampled using a digital oscilloscope, and stored on any normal PC for observation and analysis. Loss of data due to both lack of feasibility to capture GHz frequencies (power fluctuations are related to events occurring in nanoseconds), and the presence of noise is inevitable during this process

However, assuming noise is randomly distributed along the signal, it can be reduced by calculating the mean value of multiple execution rounds of the same operation. The

amount of noise present in the final mean value trace will be averaged out and effectively its effect eliminated if enough number of traces are used. It is clear here that the more noise present in the signal, the more traces are needed to get a usable signal in this type of attack.

Simple Power Analysis (SPA):

SPA or SEMA is said to be possible if observing one or a very low number of power traces is enough to extract secret key information. This is only possible if there is a direct link between the key used and observable changes in the signal, or at least an indirect link via a sequence of operations linked to the key which in turn can be mapped to the signal, such as cache hits and misses.

Here one or few traces can mean one or more clean traces, from which noise was removed using the method described above of taking the mean value of multiple execution rounds. However, there is some limitation to this technique, where alignment is needed to allow for averaging and clearly no time variations within signals should take place as well.

Differential Power Analysis (DPA):

In the absence of clear visual characteristics which can be used to directly infer secret key information, or even due to lack of adequate knowledge of the attacked device, we need to focus on data dependent power consumption at fixed point(s) of time, and use statistical techniques to correlate actual power consumption traces with simulated power consumption traces at these point(s) of time where both process the same intermediate information. Here the shared intermediate state causing the correlation implies that the key guessed (leading to this state) is correct.

Hence, we need a simulator, which is based on the most accurate power consumption model available to the attacker, to try and mimic the actual execution power traces to the highest possible degree; thereby maximizing correlation with the least amount of traces.

Power models that can be used to build a 'simulator' for power consumption include:

Hamming Difference (HD): As explained before, the total IC power consumption can rely heavily on dynamic power consumption. The HD model utilizes the relation between the number of gates flipping (hamming difference) to simulate power consumption at a given point of time. However, some knowledge of the inner workings of the IC is required, with regards to the consecutive sets of internal values during an actual transition to build a good model. This type of detailed information however, is not typically available to an attacker.

Hamming Weight: Although the name might suggest some relationship to static power consumption, this model is actually dependent again on the power consumed during the transition to and from a state with a given number of bits (hamming weight), utilizing a

statistical relationship between a middle point hamming weight and hamming difference associated with moving into and out from that particular hamming weight. As an example if we consider the hamming weight of bit string “1111 1111 1111 1111”; it is unlikely that the hamming difference in the operations leading to and from this value is equal to 0. Having non uniform distribution in hamming difference associated with a given hamming weight can potentially make this an adequate model.

Bit Model: The bit model can be perceived as the hamming weight of a single bit location, which can only be 0 or 1.

Now that we have our simulator, the steps for a DPA attack, as described in [1] are as follows:

We start by choosing an intermediate result that we can reach, using a list of possible candidate keys. If we consider Kocher’s DPA attack on DES described in [2], the attack utilizes a bit model, where the intermediate bit value (b) is located in the DES left hand side (LHS) value at the beginning of the 16th round (equal to the RHS at end of round 15). Here the (6-bit) candidate keys are those XORed with the 6-bit block containing b and the XOR result is inputted into the corresponding S-BOX prior to the final permutation phase. Here if the hypothesis key is incorrect, value of b calculated for a given cipher text will equal the correct b value -computed on the device- with probability approximately 1/2. Here the value of b can be represented/ calculated using a function f(d,k), where d is the cipher text, and k is a hypothesis key.

Now we measure the power consumption from the cryptographic device, while it encrypts/decrypts a set of D data blocks. Here, the attacker needs to record the list of data values which are involved in the function f(d,k). In our case, this is done by encrypting a list of plaintext values, and recording the list of resulting cipher texts (involved in function f). Here, the actual plain texts used can be discarded in Kosher’s attack, as it is not used in our selection function. Here we can keep our known data values (cipher texts), in a vector d=(d1,d2,d3,...,dD), where di in our case is the cipher text resulting from the ith encryption run. As we digitally sample power consumption at a given rate, we can record the power trace for a given run on the device for every data block di as a vector ti = (ti,1 , ti,2 , ... , ti,T), where T is the number of actual sampled power consumption values during the encryption run (T= run time in seconds x sample rate). So we end up with the vector d, along with following matrix T of dimensions DxT containing final aligned power traces from encryption runs:

t1,1	t1,2	t1,..	t1,T
t2,1	t2,2	t2,..	t2,T
t...,1	t...,2	t...,...	t...,T

tD,1	tD,2	tD,..	tD,T

Now we calculate the intermediate values using function f(d,k), for each cipher text in vector d, using each possible candidate (6-bit) key nibble k. Here we end up with a matrix V dimensioned DxK, where K is the number of candidate key nibbles and is equal to 64 (2^6) in this scenario. Entries in V can be calculated as follows:

v i,j = f(di,kj) i=1,2,...,D j=1,2,....K

Here every entry in the column representing the correct key, will correspond to the actual value involved in the power consumption at a given point of time; i.e. one of the columns in the T table.

Intermediate Value being manipulated at fixed point in time	Power consumption at fixed point in time
V1,k (where k is correct key)	T1,j (where j is a time during which corresponding intermediate value is manipulated)
V 2,k	T 2,j
V...,k	T...,j
V D,k	T D,j

Therefore, if we had a way to map those values V1, k .... V D, k on to power consumption values, we should find a statistical relationship between the mapped consumption values and the actual consumption values. It is worth noting here that there could be more that one point of time (j) during which our correct intermediate value is being manipulated.

4) Hence, as it is clear that we need to map the intermediate values onto power consumption values, we construct a new matrix H with dimensions DxK using our simulator.

where hi,j = SIMULATOR(vi,j)

Now we are ready to find statistical relationships between columns in H and columns in T. One way is to create a matrix r, to estimate to the correlation coefficient between

simulated power traces and actual power traces. Here for every hypothesis key, more than one correlation can take place where the same intermediate value is being manipulated, hence entries in R are as follows:

$$r\,i,j = \frac{\sum_{d=i}^{D} (h\,d,i - mean(h\,i)).(td,j - mean(tj))}{\sqrt{\sum_{d=1}^{D} (h\,d,i - mean(h\,i))^2 . \sum_{d=1}^{D} (t\,d,j - mean(tj))^2}}$$

Some DPA Issues

Trace alignment: As can be seen from the procedure above, it is required to have traces aligned at the point of time where the targeted intermediate value is manipulated, to find a correlation. This however is not always the case, due to undeterministic behavior of the trigger, device (such as with JAVA based systems), operating system (due for example to content switching) or insertion of random operations as means of a hiding technique. This could be overcome by either pattern matching or mounting a second order DPA attack with a huge enough number of instances, where due to a pattern present in the undeterministic behavior itself or random time imposed by dummy operations, a correlation could actually be found at all points where the target is bias towards being in certain time slots. This can only be completely averted if a purely uniform distribution can be achieved using a perfect random function. Another challenge is masking, where intermediate results are changes and thus we cannot target them using the above method. For this preprocessing is needed prior to mounting an attack.

Finally it is worth mentioning the electromagnetic emanation channel can potentially leak more information even when using a wide range EM probe [3]. This due to possible 'bad instructions' which do not result in much change in power activity [3], and due to somewhat restricted speed at which power consumption measurements can be an accurate presentation of how much is consumed instantaneously. This is because as at high speeds, power consumption from a previous instruction could lead to an elevated power measurement in a subsequent operation as there might not be a sufficient gap in between for the voltage to fall back to a near zero point prior to the next instruction starting. This makes it rather difficult to isolate power consumption related to single instructions, which is a requirement for a successful DPA as the one described above.

Historical DPA:

The DPA attacks were introduced and performed in 1999[2]. The DPA attack relies strictly on the bit model assumption, where signal points in time associated with handling a 1 bit are likely to be higher than handling a 0 bit on average. Thus, grouping signals into those associated with handling 0 and 1 bit intermediate values will lead to

that negating the sum of signal vectors in each of the those groups will result in noticeable peak(s) at the time(s) where the bit in question is being handled. Such peaks would only appear when the correct key is used to calculate the intermediate values which decide which group each signal belongs to. The main weaknesses in such an attack however is its inflexibility where only a bit model can be used, as we aim to create two groups only, which could weakly correlate to signal magnitude in comparison to a potentially more accurate byte model for example, and that the noise is accumulated in the sum; thus requiring relatively clean signals as well.

The attack was 'rejuvenated' and performed in the frequency domain, which lead to similar findings[4]. Here although the modified version had an advantage of not requiring signal alignment, it inherited the weaknesses with regards to inflexibility and requiring low noise in the frequency domain used. As such there seems to be no attempt to expand this DPA method onto more complex models –for example where we would use more than two groups and a series of calculations to see if groups rank at given points of time as they should from example according to the anticipated intermediate results' hamming weights-. One of the reasons could be susceptibility to noise accumulation and also due to that electing the correct key is rather simpler in a correlation method, where key values leading to a correlation closest to a '1' are simply elected.

1.2.2- Timing and Cache Attacks

A Cryptographic device can often take different times to process encryption with different key inputs. This is mainly due to optimization techniques, which include branching, cache hits and having non-fixed processing time for different instructions (such as division and multiplication). This key-dependant time variance can be adequate in certain scenarios to leak partial or entire key information[5-7].

In order to take a closer look into such attacks, we need to introduce an example optimization method that allows for exploiting the timing property of AES executions.

Here we can replace SubBytes, ShiftRows, and MixColumns operations with XOR operations on row entries obtained via a lookup mechanism. As means for quick illustration, a simple 'combination process' example can be described as follows:

- S-Box step: Every byte in a row is substituted, so effectively every possible row value can be substituted with another corresponding unique row value.
- The result row is shifted. Thus, we can actually map the original round input to the shifted result directly, eliminating the need for the S-Box step.
- The row is multiplied by the corresponding column of the static MixColumn table. Thus based on 1 of 4 possible different row locations we get four possible different output row values only.

Although this example is clearly non-realistic, it illustrates how mapping data -based on its value and location- to precomputed values can be done instead of performing SubBytes, ShiftRows, and MixColumns operations.

However, we need to utilize/realize that SubBytes, ShiftRows and MixColumns are in fact algebraic operations which can be combined with the input state matrix in a much more efficient matter. As described in [8-9], we need four look up tables: T0, T1, T2 and T3. Each entry in the table maps one of possible 256 values of a single 'input' byte onto a row of four 'output' bytes. For a given round i > 1 and i < 10, If we split the input state matrix Xi into 16 bytes x0, x1,...x15 and round key Ki into 16 bytes k0,k1,...,k15, computation of next round input state matrix Xi+1 can be calculated as follows:

Xi+1 = {

T0 [x0] XOR T1 [x5] XOR T2 [x10] XOR T3 [x15] XOR Concat(k0,k1,k2,k3), <-1st Row

T0 [x4] XOR T1 [x9] XOR T2 [x14] XOR T3 [x3] XOR Concat(k4,k5,k6,k7), <-2nd Row

T0 [x8] XOR T1 [x13] XOR T2 [x2] XOR T3 [x7] XOR Concat(k8,k9,k10,k11), <-3rd Row

T0 [x12] XOR T1 [x5] XOR T2 [x10] XOR T3 [x15] XOR Concat(k4,k5,k6,k7), <-4th Row

}

Now, we can somewhat attempt divide the related research into two main categories: the first assumes some knowledge about cache implementation details and access routines or the ability to access the cache memory itself [10]. The other type only has some control of what is being encrypted using the secret key, and to observe the signal [11] and time differences [5, 9] during a number of executions. If we assume a somewhat restrictive, the latter type is more likely applicable. Example of this latter attack we consider profiling time variations of a given round such as described in [9].

In this type, we organize input bytes into families based on which table they are used as index -to compute the next round-. As such, we can observe that we have four families from above: for T0: {x0,x4,x8,12}, for T1:{x5,x9,x13,x5}, etc. So clearly, a collision is very likely to take place when the values of any two in a given family are equal, as this will lead to accessing the same 4 bytes. Content switching can take place between assignment operations and potentially flush the cache memory, but such events are statistically insignificant, and furthermore exceptionally long encryption operations can be either retried or discarded.

As an example for the latter attack, we know x0 and x4 at the beginning of the second round are respectively equal to p0 XOR k0 and p4 XOR k4 => p1 XOR p4 = k0 XOR k4. Thus, under the cache collision assumption, plaintexts that have this property should take less time to encrypt on average. The linear relationship between time decline and satisfying the above equation can be observable after an adequate number of encryptions have been performed. This is required in order for the anticipated 'bias' to stand out; enough to deduce the associated key nibbles.

In such cases, a SPA-based time variation attack has a clear advantage, where only the time variation between two given rounds can be considered, and time calculations are performed with no lags due to other activities. This gives us further motivation to take a

closer look at our signals, in an attempt to find other variations and anomalies in our EM signals.

1.3- Machine Learning (ML) as an Aiding Tool:

In general ML can be defined as follows:

> "A computer program is said to **learn** from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*. "[12]

Here although template attacks seem to follow this loose definition, not much other work seems to have been done to utilize 'traditional' ML techniques in aiding SCA attacks.

As we have seen in SPA and SEMA, distinguishing operations such as add and multiply in a public cryptographic scheme and identifying cache hits and misses in optimized block cipher implementations can help deduce information about the keys used. Here, an attempt to identify add and multiply operations via neural networks has been performed to aid breaking an elliptic curve based public cryptographic scheme, this required preprocessing of the data to achieve time indifference via fast Fourier transforms followed by a PCA feature extraction technique [13].

With regards to block cipher identification, up to this moment, digital oscilloscopes do not seem to provide enough memory to record live feeds at high resolution, where attackers often utilize triggers to target very specific ranges in the obtained signal to be able to utilize its highest sampling rate provided under those restricting conditions[*]. As such, there has seemed no need to distinguish signals relating to cryptographic activity in a live feed automatically.

An attacker with high resources, however, can arrange to have a custom oscilloscope built to have much more significant memory storage allowing to 'buffer' significant periods of EM activity at a time, permitting the recording of EM activity in a realistic setting, where only a long-period trigger such as EM being above a certain threshold is to be used to start and end the recording process. These recordings need to be moved constantly to cheaper and slower persistent storage and then collected by the attacker for further analysis. In such scenarios stems a need for automatic identification of signals relating to cryptographic activity in live feeds, where storage requirements can be significantly reduced.

The extraction process could also be automatically followed by a series of template attacks that relate to certain device profiles and that for example only rely on key dependant operations such as key scheduling. Here, once keys are deduced all related signal recordings can be discarded. Extracted keys can thus essentially be fed to an attacker off location with no intervention required for significant periods of time. Here, although the cost of such a setup could be very steep, the lengths an attacker could be

willing to take should not be underestimated, and the 'logistics' which can often draw the line between success and failure outside the lab environment are worth some consideration.

In this context, performance of our technique in terms of both speed and level of confidence in classifying non-repeatable sets of operations is critical, as the identification procedure would be used heavily once recording starts to filter out random signals, and due to that we should not assume we have control over what is being encrypted by the device at a given time nor the ability to repeat encryptions in a real life setup.

Amongst classifiers which seem to offer a reasonable tradeoff between accuracy and speed despite the fact their rather primitive logic construct and the strong assumptions are Naïve Bayes. Here, all variables are assumed independent and the maximal joint probability of appearing variables being of a particular class is calculated. Finally, the ratio of data being of a particular class in the training set is used as a bias. Here, steps required to calculate an instance being of a particular class is clearly directly proportional to the dimensionality of the point vectors being trained and classified. Thus, if kept to a minimal, a rapid classification process could be achieved which could be what we require in our scenario.

A means of measuring accuracy, ten-cross-validation is commonly used when we are dealing with modest amounts of training data, where ten accuracy tests are performed by first dividing the training data into ten sets. Each time, a different nine of those ten sets are used for training and the remaining set is classified. The average score of those classification processes is then calculated. However, there is s weakness in this approach where results may be affected by dependencies between the training data sets which are clearly interrelated. As measure of overall performance, a test called Receiver Operating Characteristic (ROC) curve analysis is commonly used [*].In a ROC curve the true positive rate is plotted in relation to the false positive rate for different biases, leading to a much better approximation of how the actual testing methodology accurately separates our data.

Here a test/classifier that poorly separates instances due to inadequate 'testing criteria' could mean that depending on the bias alone, a great portion of results could alter, which is highly indicative of unstable overall performance. As such, a ROC curve is a powerful tool in that it evaluates the quality of separation between instances of two classes, which is likely the most important element in the process of building any classifier, and not how it performs under certain threshold/bias parameters only. Here a 'perfect' ROC curve passes through the upper left corner leading to an area of 1 (100%) under ROC, which takes place only if data tested is perfectly separable under the classification/ testing criteria used.

Chapter 2: Blackberry Overview

2.1- Introduction

Blackberry (BB) has been developed by Research In Motion (RIM). RIM was founded by Mike Lazaridis in 1984 and is based in Waterloo, Ontario with international subsidiaries in North America, Europe the Asia Pacific [14], and with a portfolio mainly focused on corporate based communication solutions. RIM also has a strong relationship with Waterloo University, where RIM funds some of its research institutions [15] and where Lazaridis has been chancellor of the University since 2003 [16].

The BB solution has two main consumer bases: corporate/government based and non-corporate based consumers for which BB respectively provides an enterprise service utilizing a BB Enterprise Server (BES), and the BB Internet Service (BIS). BES based solutions leverage high penetration and coverage of existing GSM networks [17], to provide secure data communication between BB mobile devices and the corporate network via "End-to-End" security (E2E) [18].

BB's E2E shares similarities to a virtual private network where it essentially provides a secure tunnel between BB mobile devices and BES. The information traveling over the network is encrypted using long and short term symmetric AES/3DES keys [18]. For email, the symmetric keys are only shared between supported email servers and the BB mobile device [18]. Other applications can leverage E2E and add an additional SSL/TSL layer for application level communication, required for other corporate applications such as a customer relationship management solution [19].

Encryption routines and key/data storage are thus an integral part of a BB mobile device. Here, BB has also devised a content protection (CP) scheme to protect E2E symmetric keys, and data stored on the device [20] which would otherwise be directly exposed if permanent flash memory was compromised, undermining the entire security model. BB also provides a Java Development Environment (JDE) [21] which allows developers to leverage BB encryption routines and device specific functionality in their applications [22].

Other features of BES setup includes setting enforceable IT policies which can be provision onto BB devices directly from a BES[23]. This allows for some control and constraint on certain aspect of employee usage of the device, which might compromise its security, such as downloading third party applications, and limiting access to domains outside the corporate network [24].

In this chapter, we wish to investigate certain aspects of the BB solution to give information about how keys are used and stored, the BB running and development environments, and some aspects of malware design, aimed to give some insight and help plan a successful SCA on a BB in a realistic setting.

2.2- Encryption Keys:

BB incorporates the use of a number symmetric and asymmetric encryption schemes as part of device activation [25], symmetric key negotiation [26], data exchange [18] and key/data storage [27]. In this section, an overview is provided to understand the security relationship between user identity, encryption keys and data.

2.2.1- Activation Keys (AK)s and Master Keys (MK)s:

During the enrollment of a new BB device, an activation process takes place, where permanent public AKs are generated with the utilization of a common password. AKs allow for authenticated negotiation of long term symmetric AES/3DES MKs which are shared between BES and BB devices to provide "end-to-end" security.

Initial MK negotiation is performed using Simple Password Exponential Key Exchange SPEKE [25], which can be generally outlined as follows (considering operation in group F*p):

- IT administrator creates a password typically between 4-8 characters linked to the user's email address, and communicates it to the user via phone or email.
- User logs in using his/her email address and password provided. Here a safe prime number p can be negotiated over the insecure channel if not predetermined.
- On both BES and BB mobile device, the secret password is used to deterministically calculate a shared generator g.
- BES and BB mobile device select secret random values a and b respectively and generate their public AKs $A = g^a \bmod p$ and $B = g^b \bmod p$ respectively.
- As in Diffie-Hellman (DH) key exchange protocol, AKs are communicated over the insecure channel, and a shared key is derived at both ends $(A)^b = g^{ab} = g^{ba} = (B)^a$
- Shared key derived can be hashed using a suitable hash function to obtain ***a*** symmetric MK of the required size.
  A simple challenge response mechanism is used to verify that both ends possess the same MK. Failing the challenge would most likely imply g was generated using an incorrect password.

At a later stage (typically at set intervals following an organization's IT policy) [27], MKs are re-negotiated in a key 'rollover' process which can be triggered by either BES or BB device. The process utilizes the permanent AKs, and knowledge of a and b secret values to generate a new MKs via MQV [26], which can be generally outlined as follows:

- Either BES or BB device trigger MK rollover process.
- BES and BB device generate and communicate ephemeral DH public keys $X = g^x$ and $Y = g^y$ respectively.
- At both ends values $d = 2L + X \bmod 2L$ and $e = 2L + Y \bmod 2L$ are derived where $L = \lfloor \varphi(P) / 2 \rfloor$ where $\varphi(P)$ is the order of the group

- Both compute shared value $V = g^{(x+ad)(y+be)}$ where BES computes $(YB^e)^{x+ad} = (g^y g^{be})^{x+ad} = g^{(y+eb)(x+ad)} = g^{(x+ad)(y+be)} = (g^x g^{ad})^{y+be} = (XA^d)^{y+be}$ computed by BB device
- V is used to deterministically calculate a new shared MK at both ends. A challenge response mechanism is used to verify possession of new MK.

We notice that SPEKE is a zero knowledge password proof [28], where the password cannot be leaked by simply observing traffic during a protocol run. The typical password used however, is weak and a correct guess would result in successful activation of a malicious party. To mitigate this, BB allow for 5 activation attempts only. This is not immediately required for the rollover process, where even if a limited number of g values can be used as an indirect result of using a weak password, knowledge of secret value a is required for a successful MQV protocol to take place, requiring solving the discrete logarithm problem (DLP) of g^a for those g values.

MQV however does not bind a user's identity to the ephemeral public key, as does for example its adapted version HMQV [29]. In HMQV using values d = Hash(X,'BES identity') and e = Hash(Y,'BB User identity') would offer perfect binding in a random oracle model [29] and computational binding with an actual hash function where collisions exist. Instead, BB relies on the fact that the password is linked at the beginning to a user's email address in the activation step to offer some weak form of binding between email address -clearly not a user's identity- and password (PW) when a unique PW is used for each account.

A more serious security hole, however, would be unsecured storage of secret value $\boldsymbol{a}$ on the BB device. Exposing value $\boldsymbol{a}$ can result in compromising the end-to-end security model, where MKs can be determined and re-negotiated by a malicious party.

Attack outline for obtaining MKs with knowledge of a:

- BES activation public key linked to user's email had already been obtained by observing traffic during SPEKE protocol run.
- Value a is recovered from BB's flash memory, where a malicious party gains temporary access to a locked device.
- Initial MK can be derived from $(B)^a$ which can be calculated by a malicious party.
- Later on, MQV ephemeral keys can be obtained by examining traffic.
- New MK can be derived as above with knowledge of a.

Furthermore, once $\boldsymbol{a}$ is obtained, traffic does not need to be continuously monitored, as key rollover process can be initiated at any time, and can be triggered by a number of events. A simple trigger could be sending a datagram to BB device encrypted using a random key. BB device will be unable to successfully decrypt this message and will automatically attempt to renegotiate a new MK [27]. Thus a malicious party can trigger this event at a time where it can monitor the traffic (such as a during an employee's lunch break). The attack can be repeated every time MK is renewed to get a current MK.

2.2.2- Session Keys (SK)s:

For every datagram sent between BES to BB device, a new session key is derived and used to encrypt that chunk of the message [27]. The session key is an AES/3DES key, and is encrypted using the current MK and sent along with the datagram as illustrated below:

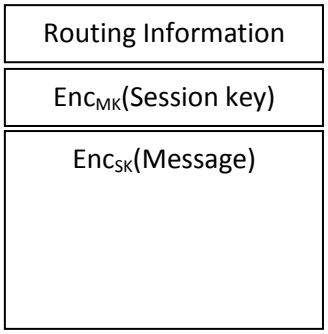


Diagram 1.0 illustrating a BB datagram between BES and BB device

Message chunks are compressed and encrypted using their corresponding session keys in cipher block chaining (CBC) mode [27]. From a SCA prospective, an attacker monitoring CPU EM activity during AES/3DES encryption will thus further require knowledge of the cipher text being transmitted as well as the plaintext to validate his/her key hypothesis during a chosen plaintext attack to extract a SK.

Plaintext knowledge can be gained by emailing the attacked device or having malware sending emails of known plaintext, which will respectively trigger decryption/encryption operations at the BB device. Knowledge of the cipher text on the other hand requires obtaining the encrypted datagram at some stage of its transmission over the insecure network and mapping it to the corresponding plaintext. Although this adds to the complexity of the attack, it is not technically infeasible. Access to one of a number of GSM network elements such as Mobile Switching Centre MSCs [30-31] and Gateway GPRS Support Nodes (GGSN)s [32-33] for example, can allow extraction of the required information as it is being temporarily stored and transferred.

After a sufficient number of SKs have been extracted during a number of differential SCA runs, an attack can then be mounted to extract MKs where the SKs now represent the known plaintext required in a differential SCA attack. Knowledge of MKs will allow decrypting all forthcoming session keys, thus stripping protection from all data sent between BES and BB mobile device.

2.2.3- Grand Master Keys (GMK)s and Content Protection Keys (CPK)s:

We notice that MKs need to be kept in persistent storage, as they are continuously used throughout their lifetime. As a means of optional protection, GMKs have been recently introduced by BB[27] to ensure the MKs on flash memory are kept in encrypted form.

The mechanism can be either triggered by the user or enforced by company's IT policy, to generate an AES GMK which is used to encrypt/decrypt the current MK as needed. Here the GMK is automatically generated and itself needs protection. Hence, content protection needs to be enabled as a prerequisite to this process.

"Content protection" is a process that utilizes a set of symmetric/asymmetric CPKs to protect user's data stored on the mobile device. As per [27], CPKs are as follows:

- A semi-permanent AES CPK to encrypt user's data including emails and calendar items. This key is also used to protect GMKs when MK content protection is enabled. The semi-permanent key is generated by the device once content protection is enabled, and is used to encrypt the bulk of data stored on a BB upon first locking the device.
- An ephemeral AES key, which is deterministically derived from the user's password. This is primarily used to encrypt the semi-permanent key when a device is locked.
- As there is no access to the semi-permanent CPK during the device being locked (as it is encrypted), an ECC public/private key pair is used to secure the data received during this period. When a device is locked, the private key is kept in flash memory in encrypted form using the ephemeral key (which illustrates the latter's second purpose), and the public key is kept in main memory to encrypt all incoming 'user content'. When device is unlocked, the ephemeral key is re-derived and is used to decrypt the private key and the semi-permanent CPK. The private key is then used to decrypt all data received; allowing user's access to the data. Finally, the data is encrypted using the semi-permanent CPK when the device is next locked.

2.3- Actual Entropy:

Passwords used for content protection are user defined and are input via BB keyboard (where US/UK keyboards are typically comprised of case sensitive English characters, digits and a set of 17 symbols). BB provides 'Normal' or 'Strong' protection options which require a password to be of length 10 and 14 respectively. A perfectly random 'Strong' password would thus have entropy of approximately 89 bits, where number of possible passwords is $83^{14} \approx 2^{6.375 \times 14} \approx 2^{89}$ (all with equal probability). Given the limitation of known cryptanalysis attacks on AES/ECC cryptography, this is clearly orders of magnitude less than that implied by using 256-bit AES or 160-bit ECC.

Although the ephemeral AES key - deterministically derived from the password - is not immediately used to encrypt user's secret information, we are able to draw the below dependency diagram, which essentially shows a 'domino' effect in exposing secret information:

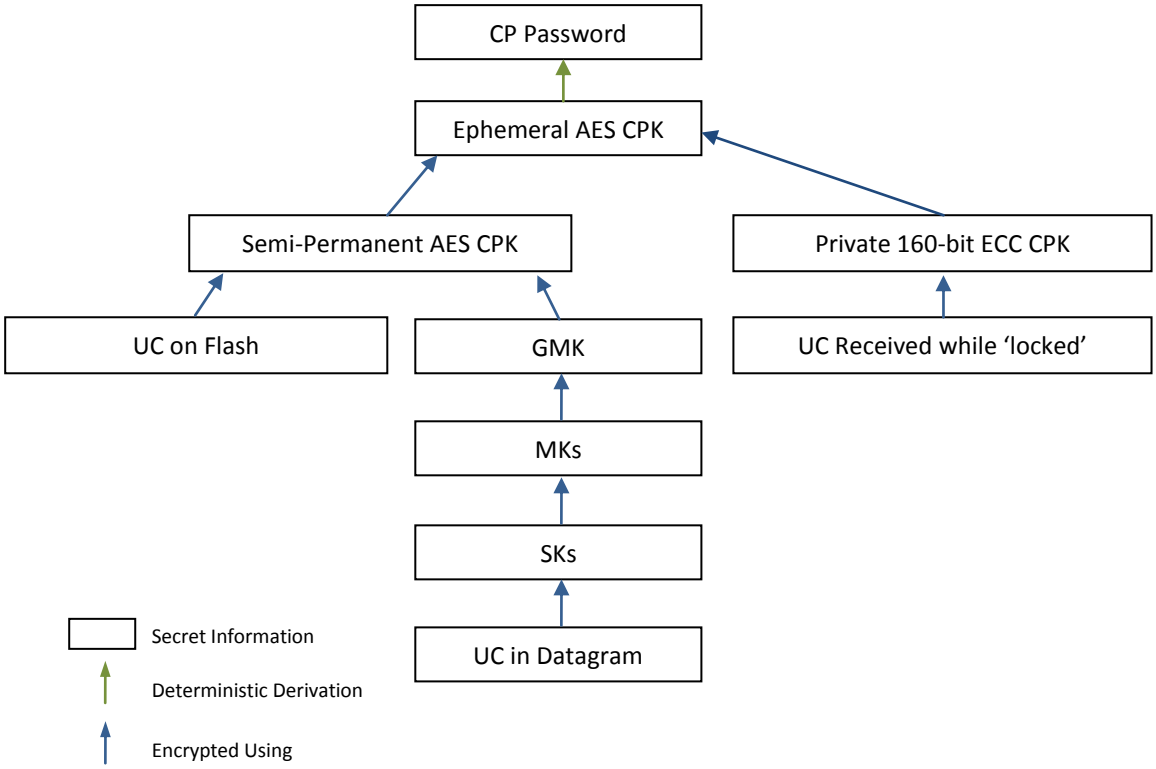


Diagram 2.0 illustrating secrecy dependencies in the BB content protection scheme

We notice the secrecy of all keys in the scheme and subsequently the data they protect can be completely compromised -either directly or indirectly- by the sole knowledge of the password. As a result, this caps the entropy of BB content protection scheme to that of the user's password (approximately 89 bits). Moreover, the assumption of using completely random passwords is unlikely to hold, as users may tend to include English words and phrases; heavily reducing the scheme's actual entropy further.

The problem clearly stems from the combination of two factors: First, the ephemeral AES key used to encrypt other keys is deterministically calculated from the password; alone leading to failing CCA2. Second, the limitation on password trials is implemented on the software level. As a result, a malicious party can perform either a brute force or dictionary attack offline, on a mere copy of the flash memory.

Therefore, it is required to break the deterministic bond between password and ephemeral key and disallow offline verification of passwords, to respectively hinder dictionary and brute force attacks on the password. Limitless offline verification can be hindered by introducing an interactive procedure involving either an online server or secure hardware which 'releases' the -non deterministically linked- ephemeral AES key only upon proving the possession of PW, with a limited number of allowed attempts. Let us describe a rather simplistic approach to do so, utilizing existing BB-used protocols and procedures:

Ephemeral key generation step:

1- Employee securely communicates CP password to BES server within corporate firewall. The CP password is stored on BES as would an activation password.

2- BES randomly generates an AES 256 key -using its SK generating algorithm-, links it to user's account and stores it in a similar fashion to that of user's MK.

PW verification and ephemeral key release:

1- Using SPEKE, a shared symmetric key is agreed upon when user enters his/her CP PW, with a limited number of trials; exactly like in BB device activation.

2- Successful key agreement will trigger the 'release' the AES key which is encrypted using agreed symmetric key and communicated back to the user.

3- BB device decrypts the AES key and uses it in an identical way to that of the AES ephemeral CP key.

Ephemeral Key Renewal:

1- Either BES or BB device can trigger CP PW renewal process, as in a MK rollover

2- User is prompted to enter existing password, which will allow obtaining current AES key. The AES key is kept in main memory temporarily.

3- With knowledge of ***a,*** use MQV and proceed exactly as in rollover process to negotiate new ephemeral AES key.

4- Stored CPKs are decrypted using old AES key, encrypted using new AES key and stored back on flash memory.

The idea of online verification itself is plausible in that it allows further control in employees accessing sensitive corporate data stored on their BBs which can be halted upon termination, but it would disallow access to UC in areas where there is no GSM connectivity. Thus, for existing BB devices in the market -where adding a HW protection chip to safeguard a random AES key is unfeasible- it would be sensible not to replace the current setup, but to add an additional enforceable IT policy for online CP password verification. But still remains an important question: does the above protocol actually 'solve' our problem, or simply move it to the domain of securing information saved on a BES?

It might be noticeable that the above protocol has been deliberately designed to highlight/exaggerate security holes in the BB solution as well, once we step inside the corporate firewall as elaborated below:

Potential Weakness	Magnitude of Problem in BB Solution	Magnitude of Problem in Interactive PW check
PW Protection on BES	During the window of device activation, leaked PW can be used to activate a BB device and hence receive/send sensitive/fraudulent data.	Time window much bigger. User can masquerade BB user at any time to obtain ephemeral key, and decrypt user content on flash
MK/AES Key protection on BES	Decrypt all messages sent throughout a MK's lifetime	Decrypt entire content on BB device during our AES key lifetime; probably containing information sent back and forth throughout several MK lifetimes.
Key generation process for SKs and AES Keys	Attacks on PRNG can be used to reduce entropy of SK. Revealing an SK will expose a single packet of data	Revealing AES key will indirectly expose all information stored on BB device.
Knowledge of private key value $\boldsymbol{a}$	As shown before, a fresh MK is potentially obtainable at any time, potentially exposing all future data	Access to a fresh AES key will expose all private information stored on the device

It is not a new concept to essentially exaggerate potential risks when it comes to establishing notions of security. For example, although some notions such as CCA2 might seem 'paranoid' to a casual listener, it is sensible to assume that attackers have more leverage than ordinarily expected; if only to compensate for the effect of fully exposing a seemingly insignificant security threat such as reducing the entropy of a single SK. By exaggerating existing potential risks in our solution, we may hope increase the likelihood of non-technical decision makers contemplating the "realness of the threat" of exposing weaknesses in the building blocks of the BB solution, which may trigger demand for a proper security evaluation of the corporate side of the BB solution.

Thus, we would probably need to perform more fundamental changes to address some of the problems discussed: For example, for better security we could utilize the first part of already used SPEKE protocol to only to agree on a common generator **g** value. This **g** value can be used to proceed with HMQV immediately for both initial MK agreement and all subsequent MK renewals. The reason for generating **g** in this way is to somewhat reduce the risk of trying to find collisions associated with the non-ROM version of HMQV. The most elaborate change required by BB here, is that user needs to generate a private key and inject it along with her and the company's public keys onto the BB

device (and also communicate her public key with BES). This is as the d and e values depend on ''BES Identity"= $g^{\text{BES public key}}$ and "BB User Identity"=$g^{\text{BB User's public key}}$ respectively [29] and because the private key is required for a successful HMQV protocol run [29]. Such a change although adds some inconvenience, greatly minimizes the risk of associated with centralized "activation password" storage, and provides a form of 'real' binding to all subsequent communication rather than the illusion of binding in the case of the existing BB setup.

To conclude, we notice that even through a rather 'preliminary' analysis that is based on BB's -severely lacking in implementation details- online documentation, it is made clear that RIM lean towards providing a "just enough" level of security, where focus is on how to market BB as a secure solution, rather than actually providing the implied high level of security to consumers. It is rather bold from BB (pun not intended), to claim UC is protected by 256-bit AES keys, while actual key space is capped to 89 bits in a non-existing ideal setting. It seems none of the official certificates awarded [34] seem to properly scrutinize many aspects of the BB solution, with particular neglect to what is behind the firewall, and lacking proper evaluation of the protection schemes and binding methods used.

2.4- BB Development:

2.4.1- Introduction

BB devices run a proprietary operating system (OS) [35], with an integrated variant of sun's K Virtual Machine (KVM) [36], hereafter called the BB virtual machine (BBVM). KVMs fall in towards the 'lower' end of the Java Virtual Machine (JVM) spectrum; targeting devices with limited resources such as mobile devices and PDAs [36]. The JVM spectrum is mainly divided based on different tiers of targeted computing platforms, in an attempt to offer the right balance between functionality and performance for a given class of hardware and services.

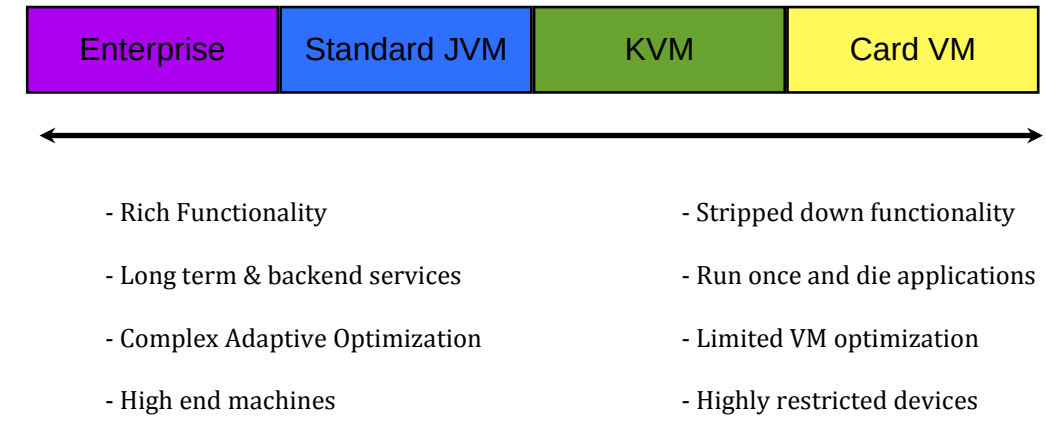


Diagram 3.0 illustrating main categories of JVMs

A J2ME environment -that runs on a KVM- is comprised of a 'configuration' which defines basic data types and operations [37], and a profile (stacked over the configuration) to provide a mobile user interface (UI), networking, persistent storage and to manage an application's lifecycle [38]. For mobile phones these are typically the Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP) [37-38]. BB is compliant with CLDC 1.1 and MIDP 2.0, which ensures compatibility with generic J2ME MIDP applications aimed at a wide range of mobile devices [39].

BB also provides its own Java application part interface (API) to allow access to a richer set of functionality and to optimized native/compiled code routines [22, 40-41]. Access to precompiled code via Java API is common in mobile environments including Symbian [42] and Android [43] -partly due to limited VM optimization- in order to limit costly interpretation of java byte code for highly used routines. Java acceleration technology 'Jazelle' [44] is also provided by ARM Central Processing Unit (CPU) fitted BB devices, to improve overall performance, by allowing direct execution of Java byte code by an on board coprocessor [44].

Generic MIDP applications (apps) can access CLDC and MIDP APIs [38] and require compulsory functions to be defined to take place when a user wishes to startApp(), pauseApp() and destroyApp() [45]. Such apps (also called MIDlets) typically run in the foreground upon explicit user action to do so and cannot leave live threads after being stopped/closed by the user [46]. However, MIDP apps offer a high degree of interpretability amongst mobile devices [38].

In contrast, BB CLDC apps have a more complex lifecycle and can be run in the background as well as the foreground. They can also be defined as system processes which automatically run upon start up and require a standard main() method [47]. BB MIDP apps on the other hand while allowing access to BB functionality, inherit the simple generic MIDP application lifecycle.

BB applications can import all three CLDC, MIDP and BB APIs [39], thus developers need take precaution not to use conflicting methods provided by both MIDP and BB APIs, such as UI APIs [39]. It is worth mentioning that BB recommends using BB CLDC applications as they provides a more flexible lifecycle and the fact that most functionality available with MIDP is also available as a more 'native feel' version with BB APIs; eliminating the need to import MIDP APIs in BB CLDC apps altogether [47].

BB provides two options for developers: Its own stand alone JDE [48], and a plug-in for Eclipse JDE [49]. It also provides a simulator, as an integral part of those environments, for debugging and testing [48-49]. In a JDE, Developers can specify application type being BB MIDP or BB CLDC. In the latter case, it can be defined as foreground, background or system background. Here we mention that as system background CLDC applications - that extend net.rim.system.Application rather than net.rim.system.UiApplication - run at system startup with no user interaction required [50], no icon appears in the menu. Thus, users could potentially be running unwanted or malicious applications without their knowledge.

We can attempt to visualize the BB environment as follows:

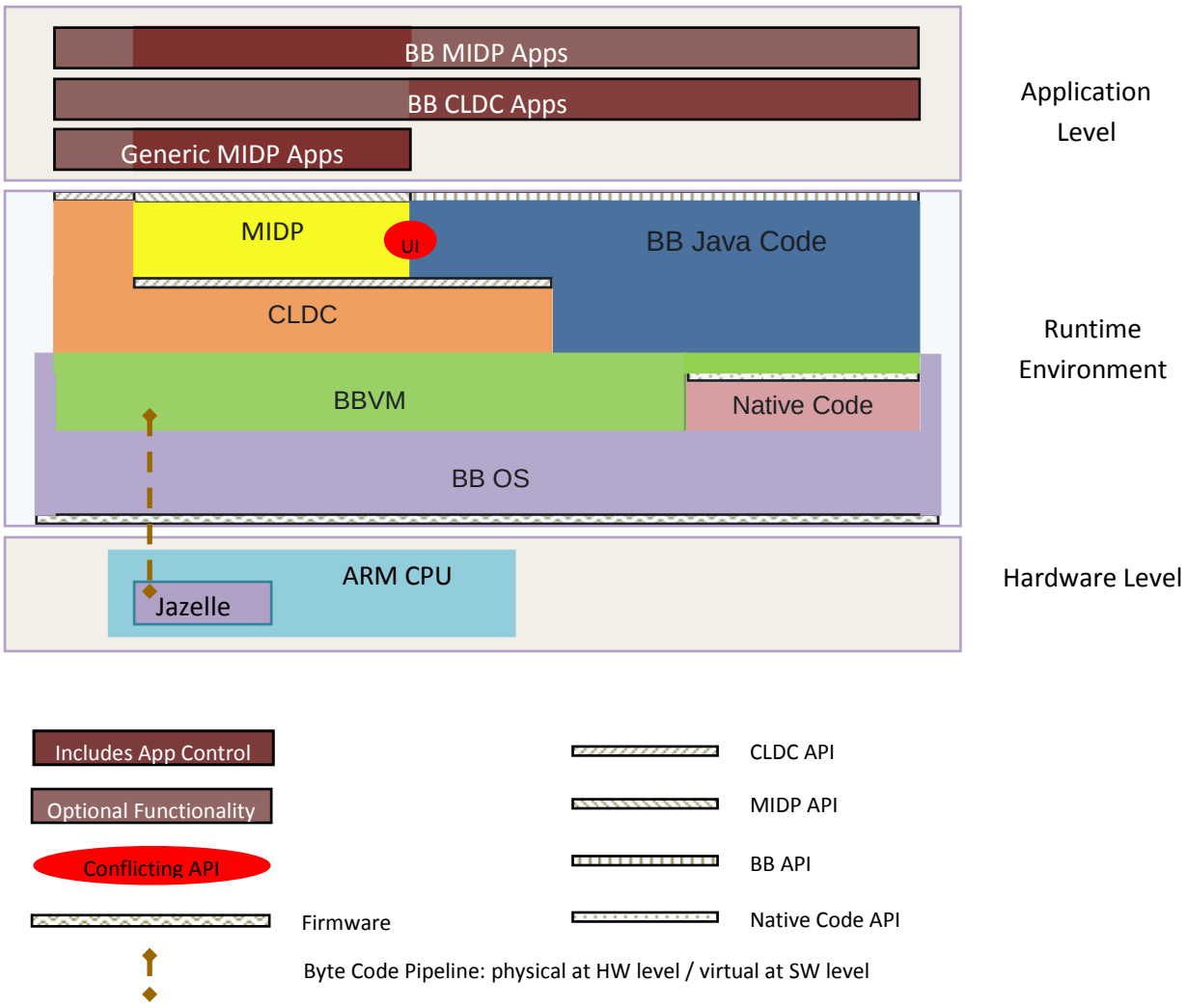


Diagram 4.0 illustrating visualization of the BB mobile device environment

In BB, the BBVM and native/compiled code routines are closely tied to the OS [41, 50]. BB calls its cryptographic routines –in the callable native routines portion of the OS- its "cryptographic kernel" [34, 41], which are exposed via BB API [51]. This allows developers to leverage existing 'optimized' cryptographic routines to provide the required functionality in their applications.

The cryptographic kernel is also part of what is considered "sensitive functionality" exposed by BB "controlled APIs" [40]. Such functionality also include accessing/ manipulating emails, contact lists, calendar items, persistent storage, various HW communication channels, SMS and key logging [40]. We may notice such functionality is considered 'sensitive' partly as it provides building blocks for all sorts of possible malware; aiming at data theft, data corruption and denial of service (DOS).

As such, BB has devised a 'code signing' [18, 40] mechanism as some weak means of non repudiation, aimed at allowing RIM to track authors of any discovered malware, and to

somewhat control and track the usage of its APIs . Controlled APIs are divided to subcategories/functionality types; each requiring a developer to possess a key/license for that category, which can be summarized as follows:

License	Category / Functionality
RIM Runtime APIs	Required for running background and system processes
Blackberry APIs	Controlling the vast majority of BB device-specific functionality such as: - Enumerating contact lists and access to email and SMS and personal data - Controlling various hardware features such as manipulating LED, screen back light and speakers - Making/Receiving Http requests - Access to low level communication channels such as Bluetooth, Wifi and USB
Cryptography APIs	BB's cryptographic library includes symmetric (block and stream) and asymmetric encryption APIs, in addition to numerous hash functions and PRNGs
Certicom Cryptography	Prior to BB's acquisition of Certicom, a separate license was required to access ECC cryptography and some public key scheme APIs.

The process of obtaining and signing applications can be outlined as follows:

2.4.2- Registration:

- Developer requests signing keys from BB's website, entering his/her personal information, credit card details and a 10 digit personal identification number (PIN)
- BB responds with three emails containing three java based 'activation' files that correspond to the three types of APIs
- Developer opens either file to start activation process.
- User is prompted to enter a password (PK-PW).
- User is then requested to move his/her mouse to generate random data needed for creation of a X501 public/private key pair (type obtained by decompiling BB's application signer, and getting error message from one of the files)
- Now, the user's public key generated needs to be communicated back to BB, during which the developer must enter the PIN used during registration as validation of identity. The public key is now linked to user's profile at BB's side.

- Developer's private key is stored in encrypted form using the PK-PW entered earlier. In addition, the information required for contacting BB signing servers for each type of API -obtained during the activation process- is saved in a .db file

2.4.3- Compiling and Signing:

- Compiling via JDE or Eclipse plug-in will generate a BB java compiled .cod file and files containing SHA1 hash of the compiled code and a list of required signatures.
- A java based "application signer" is provided as part of JDE. This prompts for user's password to release the private key, checks for list of required signatures, and for each sends an HTTP request to the corresponding signing server (from the .db file). The request is constructed as follows [52]:

    1) User's identity

    2) Hash of the compiled file

    3) Request is signed using developer's X501 private key to validate request's legitimacy

- BB signing server validates the request and checks that the user is allowed to request and receive signatures [52] (private key can be either revoked by the user or by BB upon malicious behavior)
- BB signing server responds back with a signed hash using BB's private key [52]
- The signed hash file(s) are appended to the .cod file [52], which can now be deployed and run on a BB device via a .JAD file, using BB desktop software or via USBLoader application included with the JDE.

The BBVM can validate the signatures via BB's public key embedded within the device [52]. BB can suspend a developer's signing key for a number of reasons, including publishing malicious software or malicious signing behavior by an illegitimate party. Here, there is no way to void already signed applications, and therefore BB offer a functionality that resembles a certificate blacklist, where an IT policy can be enforce a blacklist containing list of void keys of known malicious software.

2.5- DEMA Malware Design and Deployment

The aim of the attack is clear: design an application which will trigger the required encryption operations to obtain the required data for a DEMA attack, which will be used to obtain SKs and MKs from the BB mobile device. We also need to specify certain criteria in our malware; aimed to maximize odds of successful deployment and operation. There are also several approaches to incorporate social engineering into the attack, depending on how specific the target is, the extent of our resources and the nature of safeguards anticipated.

We have seen how end to end security utilizes a MK and SKs to deliver packets between BES and a BB mobile device. Data is compressed and encrypted in CBC mode, thus to obtain necessary input in a chosen plain text attack, data traveling across the network needs to be captured at some stage to correctly establish what is actually being encrypted at a given time.

As we know, end-to-end security is used for a variety of applications; however the most appealing for the least complex DEMA is using BB email functionality. This is due to not requiring additional encryption layers which are otherwise needed to communicate on an application level, beyond the BES. In addition, email communication is a basic feature available as part of BB's running environment, and can be accessed via BB APIs. The attacker here needs to figure out how it will be compressed, dissected and what is actually sent over the network. As we have mentioned before, there are also two DEMA stages before actual text can be decrypted. First, use email(s) and network data to extract an efficient number of SKs. Second, use those SKs to subsequently reveal the current MK via another DEMA attack.

Although BB restricts type of email attachments that can be received and saved, it freely allows sending any file type as email attachment. This might add a little flexibility to the attack. One main concern however, would probably be mapping 128-bit message fragments from the email sent to the exact corresponding data obtained from a compromised network element which are Xored and encrypted using an SK. Some data corruption does not greatly affect the DEMA much, as long as data alignment is maintained. This is as a fragment obtained from the network only affects the direct next Xored input computed by the attacker.

We can illustrate data input for a chosen plaintext attack in the 1st DEMA stage below:

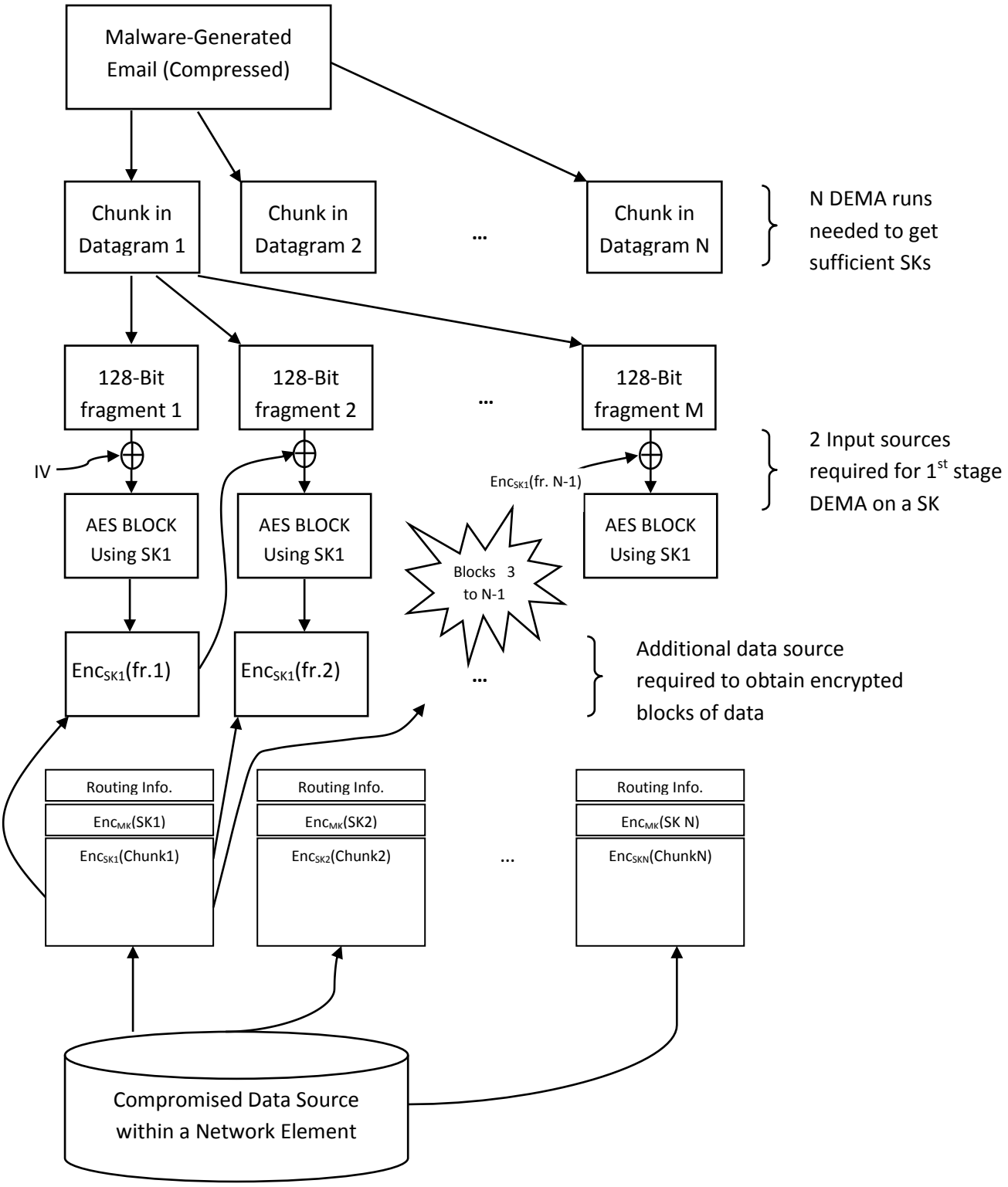


Diagram 5.0 illustrating two data sources needed to obtain actual data being encrypted in CBC mode (under assumption only DEMA is feasible)

For the 2nd stage DEMA, the attacker does not need additional effort to calculate the "chosen plaintexts" as they would simply be the SKs obtained from running multiple 1st stage DEMAs. However, during the timeframe of the 1st stage attack, EM readings need to be recorded during when the SKs where being encrypted by the current MK.

From a BB application development point of view, this is rather straight forward to implement, where triggers are set before and after sending an email containing deterministically calculated pseudo random values of sufficient length to trigger recording the required EM signals. With regards to application type, it can only realistically be a BB CLDC app defined as a background system process, where no icon appears to the user. Now all that remains is how to deploy the application onto the attacked device, and if possible set certain aspects of its behavior to minimize suspicion.

Here we need to consider the below IT policies from [53]:

IT policy rule	Description	Default setting
Disallow Third Party Application Download	Specify whether or not the BlackBerry device can download third-party Java applications. If you set this IT policy rule to True, the BlackBerry devices do not remove previously installed applications. You cannot use this IT policy rule to allow or prevent the downloading of specific applications on the BlackBerry device. For that, you must set application control policy rules.	False (The BlackBerry device allows the downloading of all third-party Java applications.)
Allow External Connections	Specify whether or not applications, including third-party Java applications, on the BlackBerry device can initiate external connections (for example, to WAP, SMS, or other public gateways).	True (The BlackBerry device allows all external connections from applications.)

Default values allow for downloading external 3rd party apps, and external communication, which allows for using a .JAD file for downloading the application, and further initiate connection with an external party. Here, if the target is specific, an email can be sent to his/her account which appears to be from a known contact with a link to the JAD file of the malicious software ("cool game?"). The user downloads the software, but nothing appears to have been installed. User might blame this on a broken link or incompatibility. Next time BB device starts however, it will seek the appropriate moment to trigger recording EM signals, and send the required email.

"Allowing external communication" can give leverage to minimize suspicious behavior. Application can interactively check for when it is appropriately positioned for EM recording; thus only producing the trigger upon close proximity. When close, it may also continue to produce the trigger until confirmation is received that positioning is ideal -

e.g. on a compromised table containing a set of EM probes- until trigger is successful. Newer devices with accelerometers can be utilized to minimize number of trigger retries [48]. Only when recording actually starts, would the application send the email. This would significantly reduce number of triggers produced and battery discharge rate which could either arouse user suspicion.

Otherwise, the complexity of the attack is increased; requiring malware to be present on user's PC which typically used alongside a BB. The malware needs download the BB app, await USB connectivity and the entering of user password to initiate a "USBLoader" command to upload the BB app. Here, attack might need to be configured/customized to run based on user's daily routine instead.

When firewall settings are enabled, user will be prompted before the malware attempts to send an email. This could hinder/alert to application's behavior. Here, if IT rules allow for "Inter-process communication" [53], app can synchronize to initiate a dummy email when another application that is expected to do so starts. This could possibly be a social network app such as Facebook [54], or CRM software. Unsuspecting user may tend to "always allow" this action, which will give a green light to the malware to operate at a later stage.

Chapter3: Lab Setup

3.1- Introduction

We have discussed how BB uses AES/3DES to encrypt OTA traffic. We also established that native code encryption functionality is exposed via BB's cryptographic APIs which are made available in its JDE and eclipse plug-in. In a previous chapter we have briefly discussed how openSCA can aid and facilitate for a DEMA, and provides means for controlling the recording of EM signals off an oscilloscope through Matlab's environment.

Thus, successfully integrating those two entities can have a significant impact on our effectiveness in mounting potential SCA attacks. In this chapter, we propose a setup which utilizes the combination of Matlab C++ code and library support and BB's existing integration tools and threading capabilities to essentially build a client server model, where Matlab can trigger any required encryption to take place for EM recording and analysis.

3.2- Components:

Oscilloscope: The Tektronix TDS 7104 Digital Phosphorous Oscilloscope was used. It can display/record signals from up to four simultaneous input sources with a sampling rate of 1GHz. As in a typical DPA/DEMA setup two channels were used, the first to capture a trigger event, while the other was used to capture/record actual EM signals. As the oscilloscope has limited buffer size, the trigger is needed to limit recording requirements by defining a timeframe in which encryption takes place. The oscilloscope is equipped with an onboard windows compatible PC; allowing installation of Windows software including Matlab and some C++ libraries, which we used to integrate with other components and to control the recording process.

EM Amplifier: EM signals are typically too weak to be observed on the oscilloscope by the naked eye. Target signals also need to be visually separable from background noise. Thus, a specialized EM amplifier is used to amplify the amplitude of a localized signal source. The Agilent 8447D was used for this purpose, and is a preamp/amp with a frequency response range of 0.1 to 13000MHZ and 25 db gain.

EM Probe and Station: A range of specialized EM probes are available at the University of Bristol, from which the one with most appropriate level of localization was elected. The election process was a time consuming one, as ARM CPU was not isolated in its own casing nor was the casing clearly distinguishable from other components on the board. The EM probe was fitted onto a station, which offers a level of control is securely positioning the EM probe.

For trigger detection, a differential probe was used, to identify both rises and drops in voltage used to respectively turn the recording process on and off.

BB Device: The BB 7290 mobile phone was used, which runs the BB 4.1 OS and the 3.80 cryptographic kernel. Here some physical preparation was needed:

1- Opening the case to expose circuitry: In our stage of a realistic attack, we first need to identify BB main processing components, to maximize the odds of a successful SEMA, where potential signal sources can be identified and where signal is less likely to be distorted beyond visual recognition, as would more likely be the case otherwise.
2- BB devices do not power on when connected through a USB power source alone, and always need to be connected to a battery during operation. Thus, having the circuit exposed; we needed to solder the BB battery contact points to the battery casing. As there were four metal contact points and the current recorded off the battery was less that carried ordinarily over a USB cable, such a cable could carry the required power over its four wires. Thus, a standard USB extension cable was used, where the end coming from the BB device could be plugged to the end connected to the casing. This offers flexibility in disconnecting power when needed and potentially connecting other BB devices later on with less effort.
3- Modify trigger points: As a differential probe was used to detect voltage rise/drop triggers, both LED and back light connectors were examined and were found to produce a sufficient identifiable rise in voltage. Here, LED connectors were elected as they have less latency after being turned on for subsequent operations to start, which reduces the required recording time and the odd of variation in encryption starting points. As such, we modified the LED connectors to allow attachment of the differential probe.

3.3- Backdoor Election:

As discussed before, BB offers a range of channels, through which an external party could use for communication, including HTTP, Bluetooth and USB. Here the most suitable backdoor is clearly a low-level USB channel. This is due to the fact it does not require any wireless communication which could generate significant amounts of EM, and due to the expected lower overhead. Time and memory overheads increase the odds of garbage collection, and time jitters and thus clearly need to be avoided when possible.

3.4- Integration Approach:

For our elected channel, BB offers a C++ library which allows for sending data between a PC and BB mobile device. This could be imported into Matlab C++ libraries and used to develop a BB USB client that works from Matlab environment. We notice that some previous attacks use the USB channel only as a trigger to start a series of predefined encryptions. This likely due to the fact that PDA development environments are well known for being notoriously challenging to deal with and as such applications flows are kept to a minimal. Nevertheless, such a setup exhibits properties of very poor design mainly due to the following:

- Lack of flexibility: As routines are predefined, either a single operation is defined per trigger, or a series of operations. In the first case, operation and/or input file

needs to be altered once EM is recorded. In the latter, no retries are possible for a single operation, and any unrecorded/missing operation means that the whole routine needs to be repeated. Odds of recording failure increase with a huger number of recordings and so does the associate time penalty for repeating a long routine; leading to a potentially very cumbersome situation.
- Added prerequisites for other team members who may wish to work further with the device. These are associated with the need to change code and/or upload files used for a chosen plaintext SCA attacks.
- Potential inconsistent/unpredictable behavior of mobile device upon issuing the trigger, as different versions of code and text files are likely to exist especially when multiple devices are being attacked by different team members.

Therefore, we have proposed a client/server model, where the full breadth of BB encryption functionality can be triggered via ASCII commands which contain the following details:

- Trigger type: LED, backlight, etc..
- Encryption algorithm: AES128 / AES192/ AES 256 / DES
- Predefining Keys to be repeatedly used in a given session
- Plaintext entered in HEX
- Time delays between trigger and encryption routines if needed

Here we notice as plaintext, keys and encryption algorithms are fully configurable via command, Matlab scripts can take full control over BB encryption operations with no modification on BB device or BB USB client. Command user documentation is in appendix A.

3.5- EM Probe election and positioning:

Main challenges associated with this task can be summed up as follows:

- Previous attacks on ARM7 board have been performed by various cryptography teams around the globe, while none have been explicitly made on a BB mobile device. This limits any room for consultation to achieve optimal probe selection and positioning.
- BB board contains a complex set of chips used to manage GSM wireless communication as well as other application specific computation.
- ARM7 chip is embedded in a casing which includes other chipsets, leaving greater room for interference, and to identify exact position of CPU within this casing.
- BB OS and a number of background process are running alongside our encryption routines, so relevant EMs are mixed with those due to other CPU activity.
- We possess no knowledge regarding implementation details of AES on BB devices, nor about BB's proprietary OS content switching properties. Thus, an AES trace could be hidden in one or possibly more chunks of noisy EM signals. Other non-deterministic java operations further complicate our task.

Thus, selection of EM probe needs to be optimal, as too big a probe can capture EMs from a number of chips embedded within the same casing, while too small a probe might not capture enough data to aid visual identification of AES rounds. Positioning of

the probe needs to be quite accurate also, as minor shifts could easily lead to missing ARM7 related activity.

Chapter 4: Analysis & Evaluation

4.1- Introduction

The integration between Matlab and BB was successful, where a token-based command can be issued to request any required encryption. The setup allowed us to record and observes AES signals in the shortest possible timeframe; allowing for optimal resolution settings to be used. Here, Matlab instructs the oscilloscope to initialize and start recording once a predefined trigger is observed, with a predefined resolution and maximal recording time. The Matlab script was also created to include a per-command retry mechanism if recording fails for any given reason.

During our acquisitions, all attempts have been successful. Those were thousands of acquisition issued throughout a number of days. During this period, the server process did not show any signs of instability, excessive battery consumption nor performance degradation. Thus, no further stress testing was deemed necessary; given that the aim of the server is performing single operations at a time to allow for observation and recording.

The recorded signals were automatically packed into openSCA 'containers', which allows for mounting an immediate attack if required. Here, a dummy attack was performed to insure no compatibility issues or unpredictable problems would surface during this process. The setup was agreed to be suitable, and the application has been handed over to the appropriate team.

As anticipated from previous work and observations experienced with Java enabled hardware and mobile devices, our recorded signals were quite misaligned and fluctuating. Under those circumstances, some efforts may lean towards frequency based attacks as they are time indifferent. However, we lose valuable timing information in fast Fourier transforms which limits the types of possible SCAs on those 'mutilated' signals. As such, we shall be adopting this approach.

In addition, some effort need be done to look into reasons for jitters and fluctuations beyond Java and OS related activity. Here, other unconsidered phenomena could also potentially affect the frequency component of the signal; undermining the effectiveness of DEMFA in certain cases as well. Finally, a high-performance and self-contained identification method is proposed which addresses issues related to limitation in available training data and high performance requirements expected in a realistic setting.

4.2- SEMA:

Discovering the hidden AES signal was the result of an extensive 'semi-educated' trial and error process, in an attempt to locate the ARM7 CPU and find the combination of most suitable probe, position, and oscilloscope settings, which finally allowed us to distinguish the EM signals generated during BB's obscure implementation of AES.

Initially several specialized probes were tried out to narrow down certain areas of high EM activity, where we started with large surface area probes, and went to smaller ones in time. With no detailed information about the BB OS, its AES implementation or the ARM7 exact placement, any fluctuations in the signal were considered as possible candidates; correlating to our goal signals. Here, even with a trigger-set timeframe, not being able to clearly distinguish the signal, adds considerable hindrance and complexity to future SCA attacks this projects aims to facilitate for; undermining its entire purpose. Finding the signal on the other hand opens a window of opportunity to further facilitate for SCA attacks in this project, where signals can be potentially aligned and identified in an efficient matter, as a step closer to a DEMA in a realistic setting.

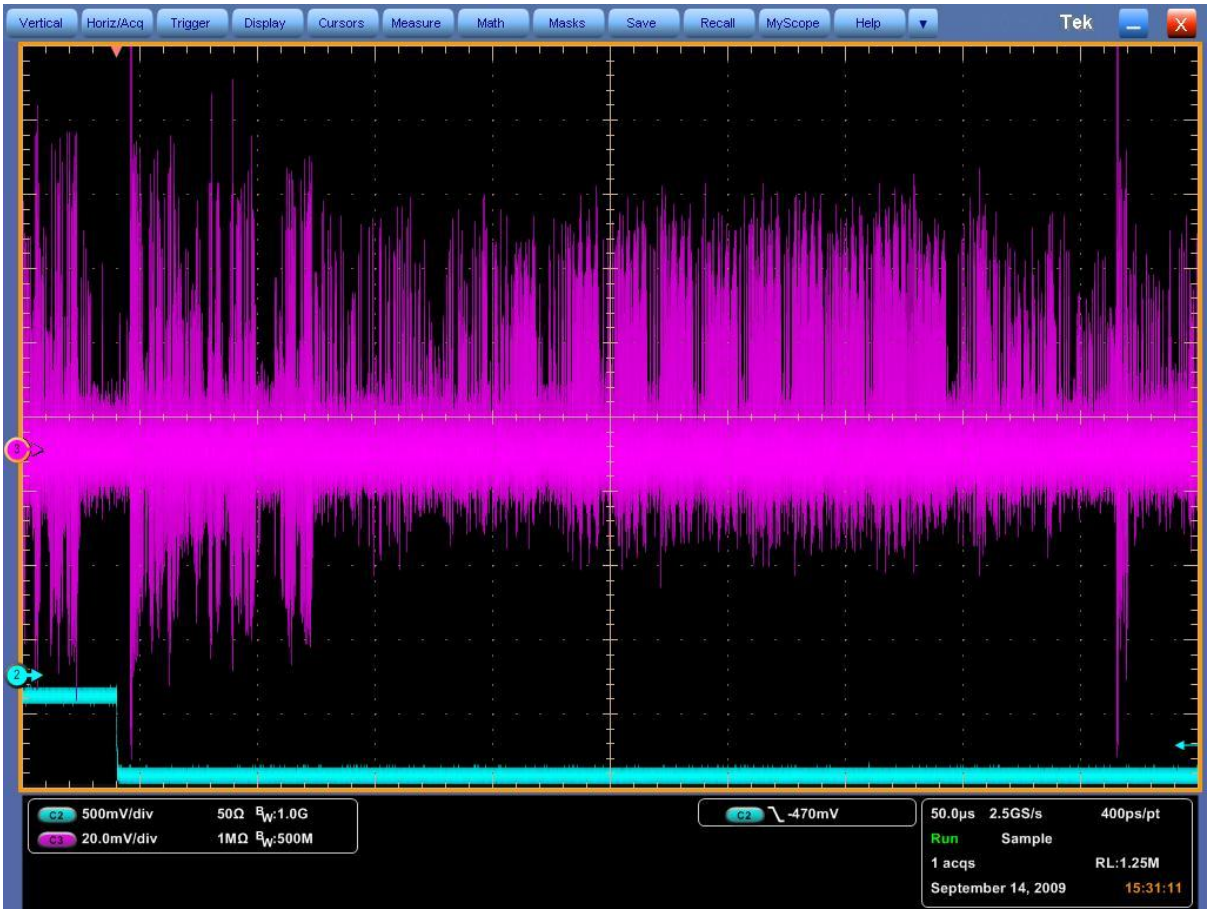


Figure 6.0: Snapshot of EM signals recorded off a Blackberry 7290 during a 128-bit AES execution

4.2.1- Initial Observations

The first observation was that all rounds were in sequence, indicating BB's OS content switching allows for their full uninterrupted execution. In addition, there seem to be no 'dummy' operations randomly inserted within the routine as mitigation against SEMA and DEMA. Adding significant dummy operations would clearly make the process of identifying the AES signal more challenging; especially if sufficient to force execution to pass through a number of content switches and CPU interrupts. With regards to DEMA, this could have served as mitigation against first order differential attacks which require perfect time alignment.

The ten rounds are also clearly separable; allowing us to notice that the first and last rounds respectively run in significantly and slightly less times in relation to the remaining rounds. The first round also seems to be more 'separated' from the following rounds. Here, as EM intensity is likely to correlate to high CPU activity, we can assume that those low activity areas are where a lookup or writing is performed on BB's flash memory, which would explain the need for the CPU to slightly 'wait' a number of cycles for data to become available from a slower data source. We notice that this wait period (separation) between rounds decreases with time as well. Finally, there is a long period of low CPU activity prior to all ten AES rounds, and we also notice four peaks in every round.

4.2.2- Content Switching:

From the snapshot above, we notice a very sharp rise in activity before and after the AES routine. This is suspected to be 'hard' content switching, where an interrupt is sent to the CPU by the OS to change the active process. This is noticed throughout our recordings, and can be explained as follows: As we have mentioned, BB encryption routines are fast precompiled routines. As such, they would not be executed on over the BB JVM. Instead, the current java process handling our request needs to switch control to another process which lies beneath the JVM to execute the optimized code. Once routine ends, control is switch back to our java process. In order to confirm that the spike is in fact related to content switching activity, we observed signals where the AES execution was clearly interrupted. Here the same spike in EM is observed in all such scenarios.

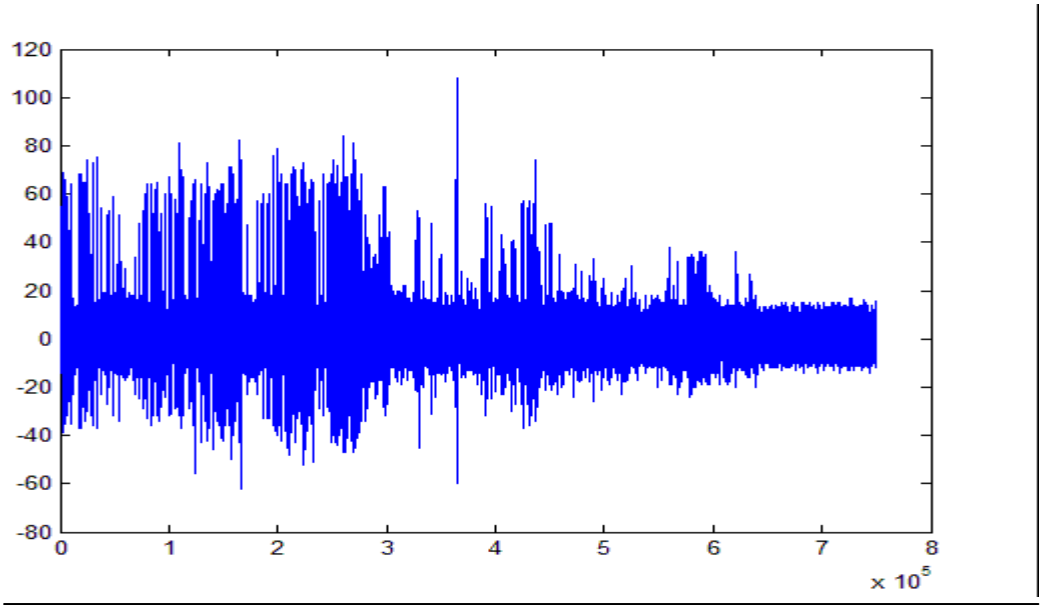


Figure 7.0 illustrating AES execution being interrupted after 3rd round

4.2.3- Jitters and Fluctuations:

Variations in 'wait' periods between consistently high CPU-activity periods/blocks can be explained by variable cache hit rates, where a set of memory access steps are followed by a set of computational steps. However we also notice other rather peculiar variations, which stand out in numerous recordings:

1- Gaps within anticipated high CPU activity periods. Some arguments against this being related to un-deterministic Java activity are as follows:

   - We have already observed how a suspected 'hard' switch is needed between native code routines and a java process. For java to take control back from the OS, content switching would take place that could be distinguished in the EM signals
   - Although Java housekeeping may appear un-deterministic as the JVM code is never disclosed to public, it would make sense that housekeeping activities that would cause soft switching to take place periodically when certain conditions are satisfied and not randomly every few cycles.
   - Java is slow compared to native code, thus relatively modest Java activity would be noticed as a significant gap in the AES pattern which would dominate the timeframe

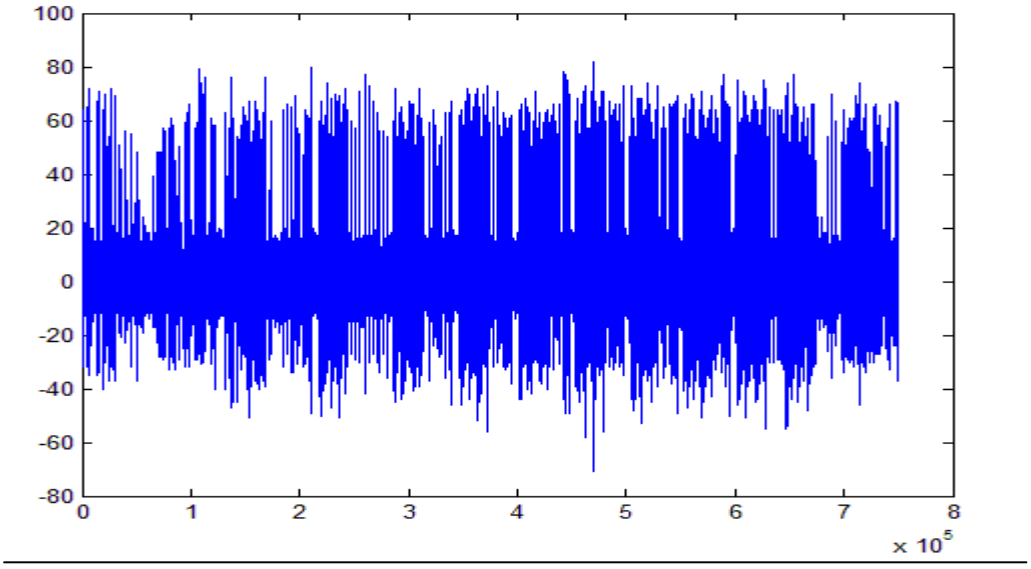


Diagram 8.0 illustrating jitters, inconsistent with JAVA housekeeping activity

2- We also notice fluctuations, where CPU speeds seem to vary within the timeframe.

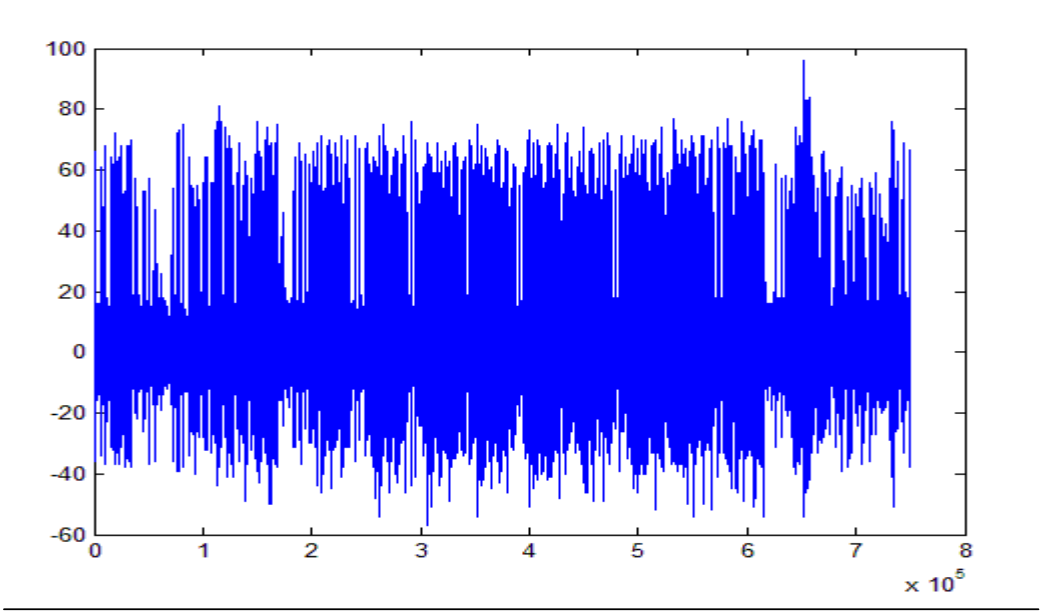


Diagram 9.0 illustrating 'analogue-like' fluctuations, where CPU seems to runs at various speeds

Such fluctuations are best explained by examining technology embedded within the hardware. CPUs targeting mobile computing platforms are specifically engineered with energy preservation as a key goal. Here, several algorithms and implementations have been developed and tailored to achieve best Million Instruction per Joule (MIPJ)[55].

This is also true with ARM, as it provides means for manipulating CPU speed and energy consumption to meet specific performance and battery life requirements. Here, manufacturers are provided with a set of interfaces that allow controlling its speed.

For example, “performance level programming” (PLP) is provided via the a configuration interface[56]. Here, an eight bits ‘programming interface’ provides support to providing 129 levels of “fractional performance” levels ranging from an idle 0% CPU utilization to 100% full performance. The extents to which those levels are actually utilized are at optional, but it is likely that mobile phones such as BBs will tend to use such technology to maximize battery life.

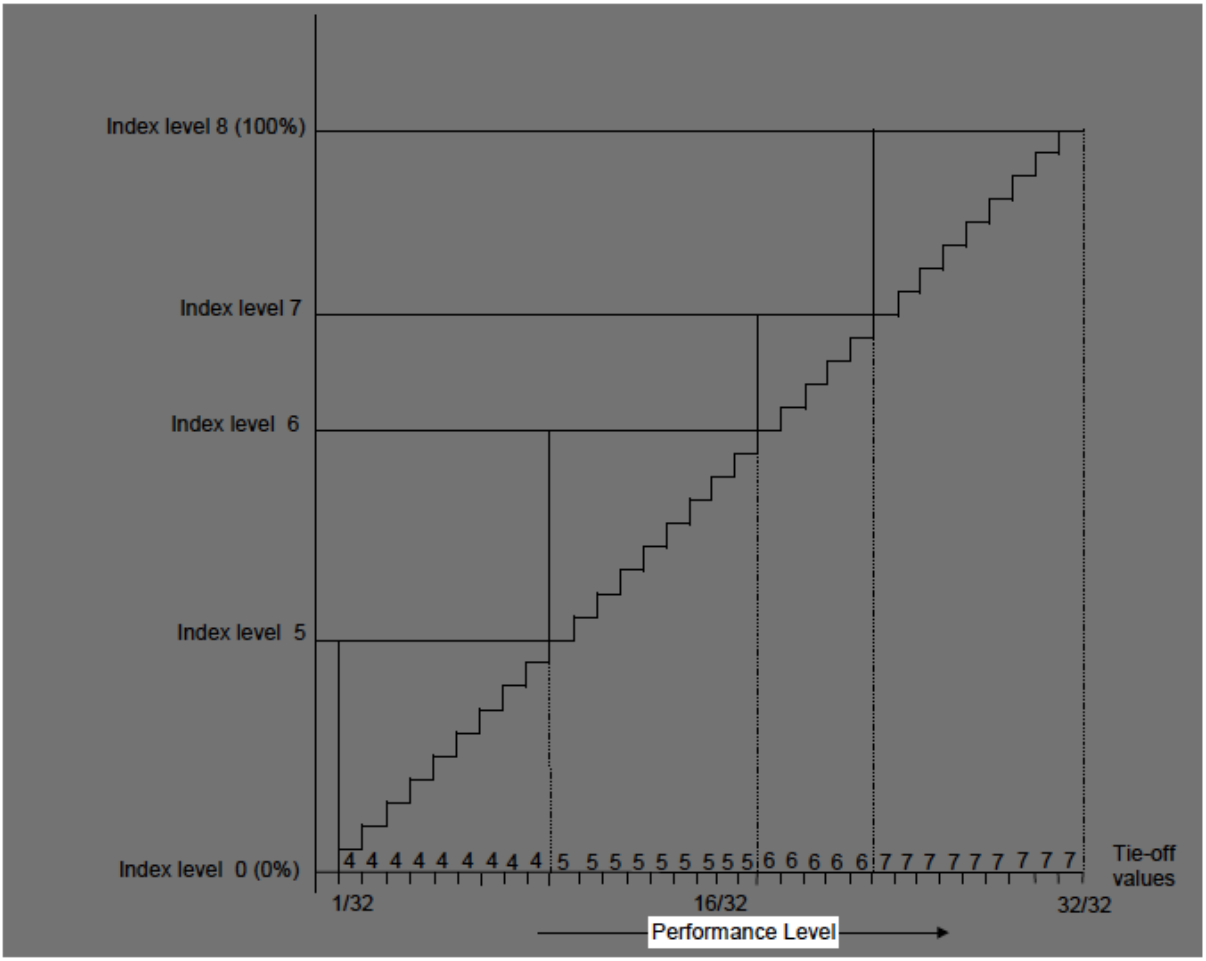


Figure 10.0 illustrating multiple levels of performance achievable via PLP interface

ARM mention that “the performance levels can be non-linear”[56]. Here, one scenario is that only clock rate is being manipulated; this is unlikely to affect the EM relating to an operation taking place within a given clock cycle. However, to prevent loss of information, oscilloscope needs to be configured to a sampling rate that allows for adequately recording EM traces at 100% CPU utilization. PLP is one possible explanation to the jittering and long total execution time seen if figure x.0, where CPU speed is set to 0 or a very low speed at those oddly placed gaps.

However with features such as "Voltage Scaling and Pulse Width Modulation" [56] speed and intensity of an actual clock pulse might be effected as well, and thereby the speed of the instruction as it is being executed. This would likely have an effect on the magnitude and frequency components of radiated EMs for instructions executed at different speeds. Here, it would be interesting to closely study the effects on an attack under those conditions, and whether simple procedures can be used to detect and 'unify' EM signal as if running at a single speed.

4.2.4- 'Simple' Reverse Engineering:

Observations in the signal can be potentially mapped to one or few of a potentially significant number of different AES implementations, where some can be ruled out easier than others. As such, we shall not aim to claim a certain implementation is used by BB. But instead, we shall attempt to rationalize our visual observations by attempting to map them to two standard implementation 'families' only, under the set of assumptions that now can be made given our previous observations. The motivation behind this is to show how as we have initially ruled out insertion of dummy operations during the encryption; attackers may further narrow down possible attacks with a certain degree of confidence. As such, we aim to illustrate how a malicious party can preliminarily attempt do so by example.

Traditional Byte-Based Implementation:

Here, the following assumptions regarding our signals are made:

- Long wait before first round is due to S-Box table loading
- Waits between high CPU intensity periods are due to S-Box table lookup operation
- Due to caching, those lookup periods decrease with time
- Although ARM is a 32-bit processor, optimized methods do not improve overall performance due to lack of sufficient memory needed for lookup based methods to yield any actual improvement.

Hence, round to high intensity distribution is as follows:

EM high-intensity block number	Associated AES Activity (AAA)
1	- First round key XOR
2-9	- S-Box value assignment (value already looked up during wait period) - Shift Rows - Mixed Columns - Add Round Key
10	- S-Box value assignment (value already looked up during wait period) - Shift Rows - Add Round Key

From the table above however, we notice the Mix columns step is not performed in the last round. As this involves matrix multiplication, it should result in a significant reduction in time; altering the pattern noticed in the last round in comparison to previous rounds. This is not noticed with our traces. On the contrary the period of high EM activity in the last is no shorter than the proceeding eight rounds, nor does it appear distinguishable.

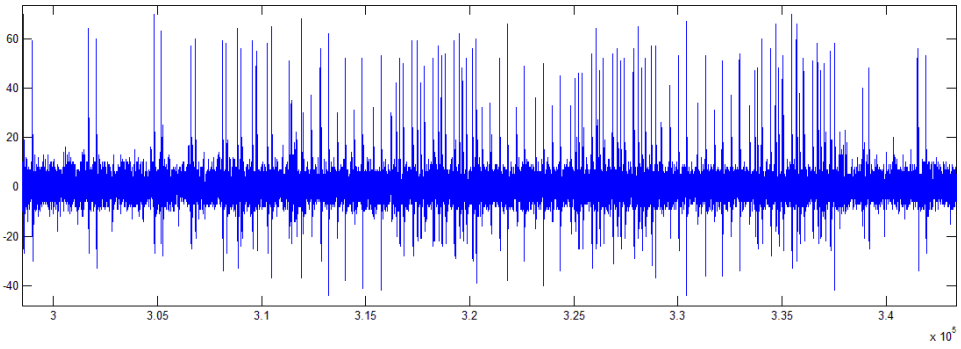


Figure 11.0 illustrating a closer look into one 'block' of EM activity mapping to the second round

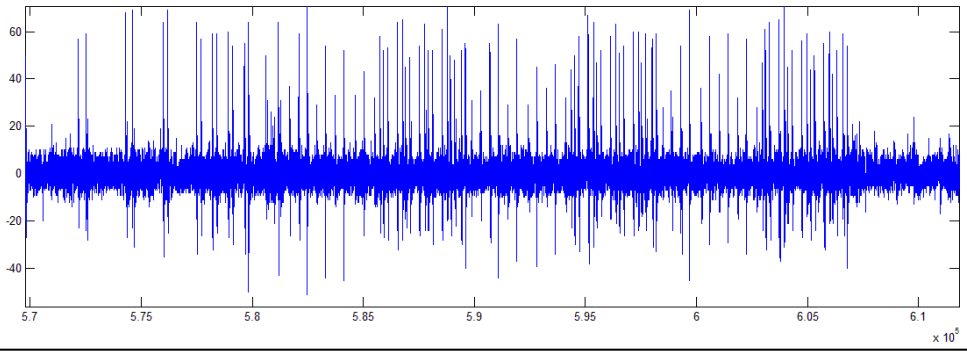


Figure 12.0 illustrating last EM block is not clearly shorter / distinguishable from previously shown block

Thus, in light of the above, we may have a high degree of confidence in eliminating a non-lookup based method. As a simple example for narrowing down possible attacks, an attacker would be unwise to use intermediate values resulting from an S-Box lookup in his/her DEMA attack, nor to obviously attempt S-Box based cache/timing attack either, where otherwise relations between plaintexts and keys that lead to accessing the same S-Box entry would have a less execution time on average given adequate encryption operations are performed; potentially allowing to deduce the key which exhibits this property the most. Instead, time can be saved in mounting other attacks that are more likely to succeed.

Optimized 32-Bit Implementation (4 table version):

We recall calculating state matrix of next round from a previous chapter:

Xi+1 = {

T0 [x0] XOR T1 [x5] XOR T2 [x10] XOR T3 [x15] XOR Concat(k0,k1,k2,k3), <-1st Row

T0 [x4] XOR T1 [x9] XOR T2 [x14] XOR T3 [x3] XOR Concat(k4,k5,k6,k7), <-2nd Row

T0 [x8] XOR T1 [x13] XOR T2 [x2] XOR T3 [x7] XOR Concat(k8,k9,k10,k11), <-3rd Row

T0 [x12] XOR T1 [x5] XOR T2 [x10] XOR T3 [x15] XOR Concat(k12,k13,k14,k15), <-4th Row

}

Here, we notice that for a given round we can abstract operations involved in a number of ways, as we can reach the same end value Xi+1 with different ordering of underlying computation:

1- Four identical blocks of EM activity: each block represents one line from above. Example:

   Xi+1 [Row 4] = T0 [x12] XOR T1 [x5] XOR T2 [x10] XOR T3 [x15] XOR KEYi [Row4]

2- Two different blocks of EM activity:
   a. 1 Block to schedule and assign all round keys values + 1 Block to retrieve table values and perform XOR operations
   b. 1 Block to retrieve all Ti[] table values + 1 Block to get perform key scheduling and XOR operations
   c. 1 Block signifying first block of both a and b and another block to perform XOR operations

SEMA for Abstraction Method 1:

Initially we have a discrepancy, where we cannot trivially divide a round to four identical sets/blocks.

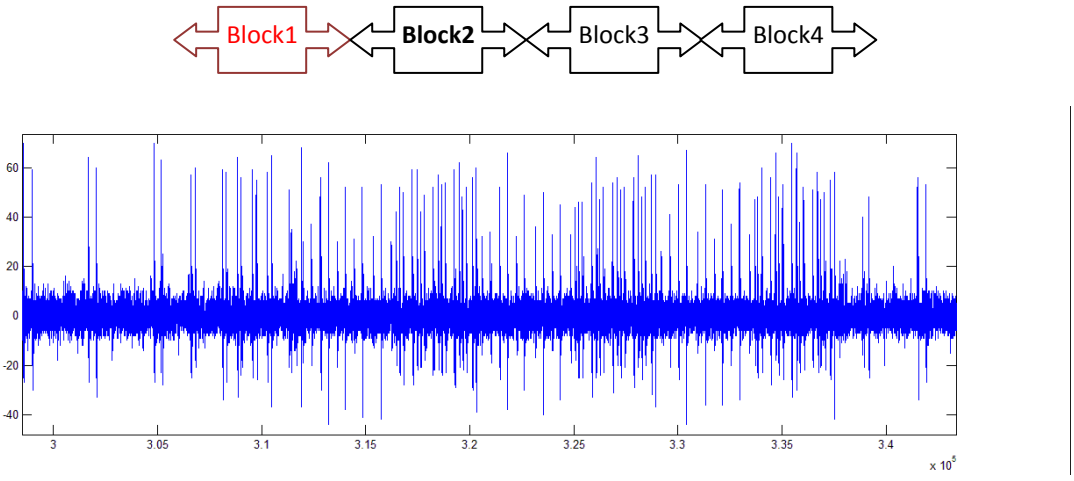


Figure 13.0 illustrating block1 is not part of four consecutive identical sets of operations

However, let us now assume BB utilize ARM's Intelligent Energy Control to reduce clock rate at starting points of every while loop step as some mitigation against DOS. This comes along side features such as where only background threads are allowed to perform infinite loops to avoid DOS.

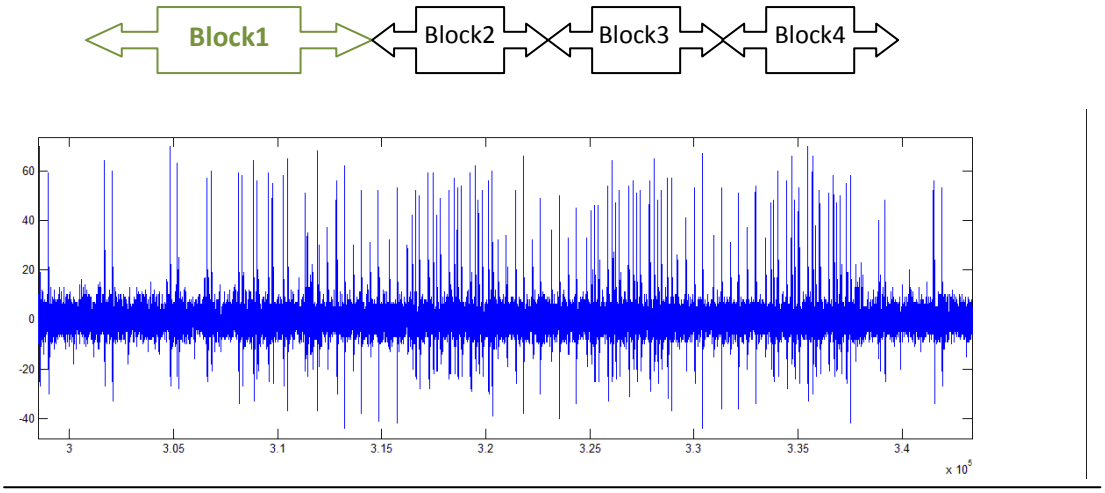


Figure 14.0 illustrating block1 now conceivable as similar to other blocks

Here it is worth noting that we are not claiming the four table lookup method is used by BB, but rather to show how a huge number of possible assumptions can be made, due to the possible variation in speed and signal fluctuations alone. In such circumstances, correct speculation regarding the signal could be easily discarded if not matched with the correct set of assumptions, while on the other hand, incorrect speculation can be validated under a technically valid but nevertheless incorrect set of assumptions.

SEMA for Abstraction Method 2:

Here, we also notice that the level of 'confidence' in our assumptions is related*to the level of abstraction made as well. As all rounds 2-10 are very similar, any division that maps two set of operations to two high activity portions/blocks becomes possible due to the poor level of abstraction. In the other far end of the spectrum, being able to logically map individual machine instructions to visual observations leads to a higher level of confidence, as it is might be unlikely to find the same number possible matches at that level of abstraction.

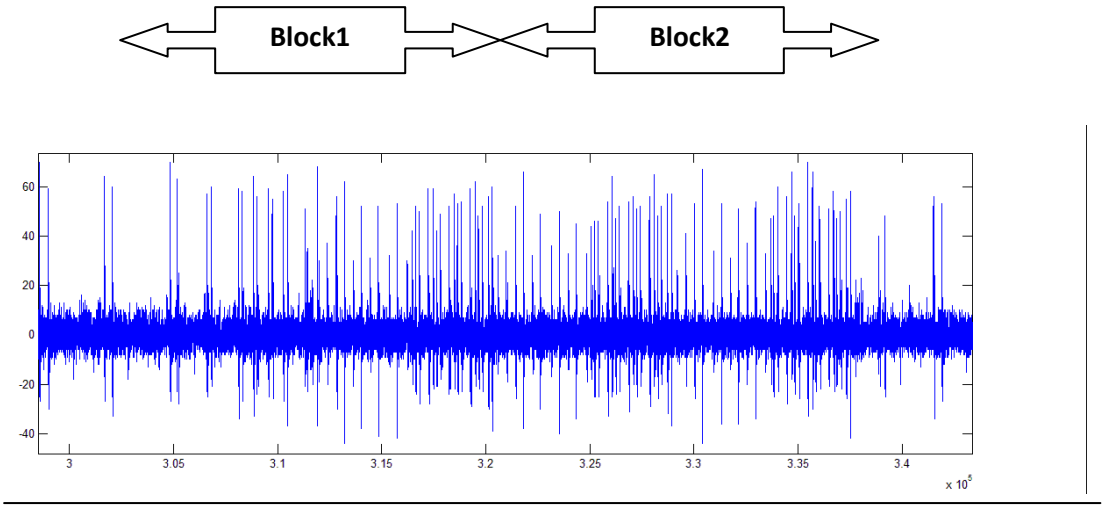


Figure 15.0 illustrating SEMA 'Procrustes': With poor level of abstraction, we can map all a,b and c

Thus, how can we achieve higher levels of confidence without dissection down to the machine instruction level; where a huge number of sequences of operations are possible for significant number of AES implementations? Here, clearly using statistically based methods are sensible to formally validate our assumptions. Such SCAs include DEMA and possibly timing attacks as well. These, however require alignment to maximize effectiveness and potential.

4.3- Signal Alignment:

Traditional alignment methods such as minimizing square difference and maximizing correlation have initially performed poorly with regards to aligning our EM signals, where interesting points have been set to areas surrounding the first round. It is worth mentioning that those alignment methods are rather expensive computationally as well with a complexity of $O(N^2)$, where N is the length of our interesting points. This is as for every possible position, a constant multiple of N operations are required to calculate square difference/correlation. Poor results can be argued to be due to EMs' non-linearity in comparison to electric power consumption; thus acting as a natural mitigation against traditional alignment methods, where we effectively have 'teeth' - opposed to pits slopes- that could lock at one of multiple locations leading to the noticed false positive alignments.

Thus, some adaptation was needed, where two anchors were found that were placed near where the AES "While" statement is expected to initialize. Their presence, location relating to the first round, fixed separation distance and lack of intermediate signals above a certain threshold all contributed to their uniqueness, and the speed to which they aligned a given signal. Here, it is worth mentioning that the theoretical complexity here is also $O(N^2)$ as we need to check intermediate values for every two anchors satisfying magnitude threshold and separation criteria, however, in practice any two other anchors were immediately illuminated due to them being in high CPU activity

areas; immediately failing the other test. Thus, alignment is quick, and can be either incorporated into a DEMA attack, or within the recording process itself.

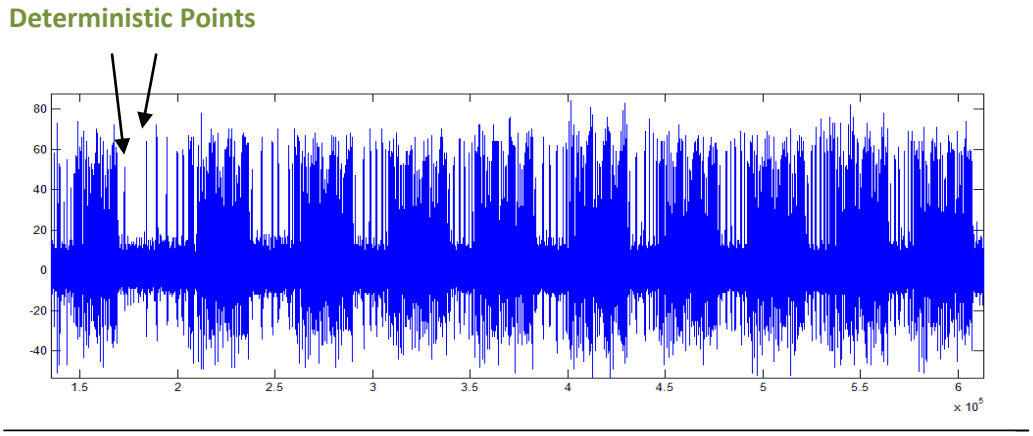


Figure 16.0 illustrating points suspected to be related to while loop initialization

Here, having found while loop related activity would be of significance, as there is less reason for variable separation between the end of the first round, and the while loop starting; due to lack of memory access operations in between where cache hits can alter timing. Moreover, even if there were variation within the first round, aligning in this way would keep variations between accessing the same text or key portion for calculating the next round state matrix to a minimum between recordings; thus, allowing for initially minimizing the needed number of additional traces need to find correlations, and if not aiding for potential automated 'unifying' of signals.

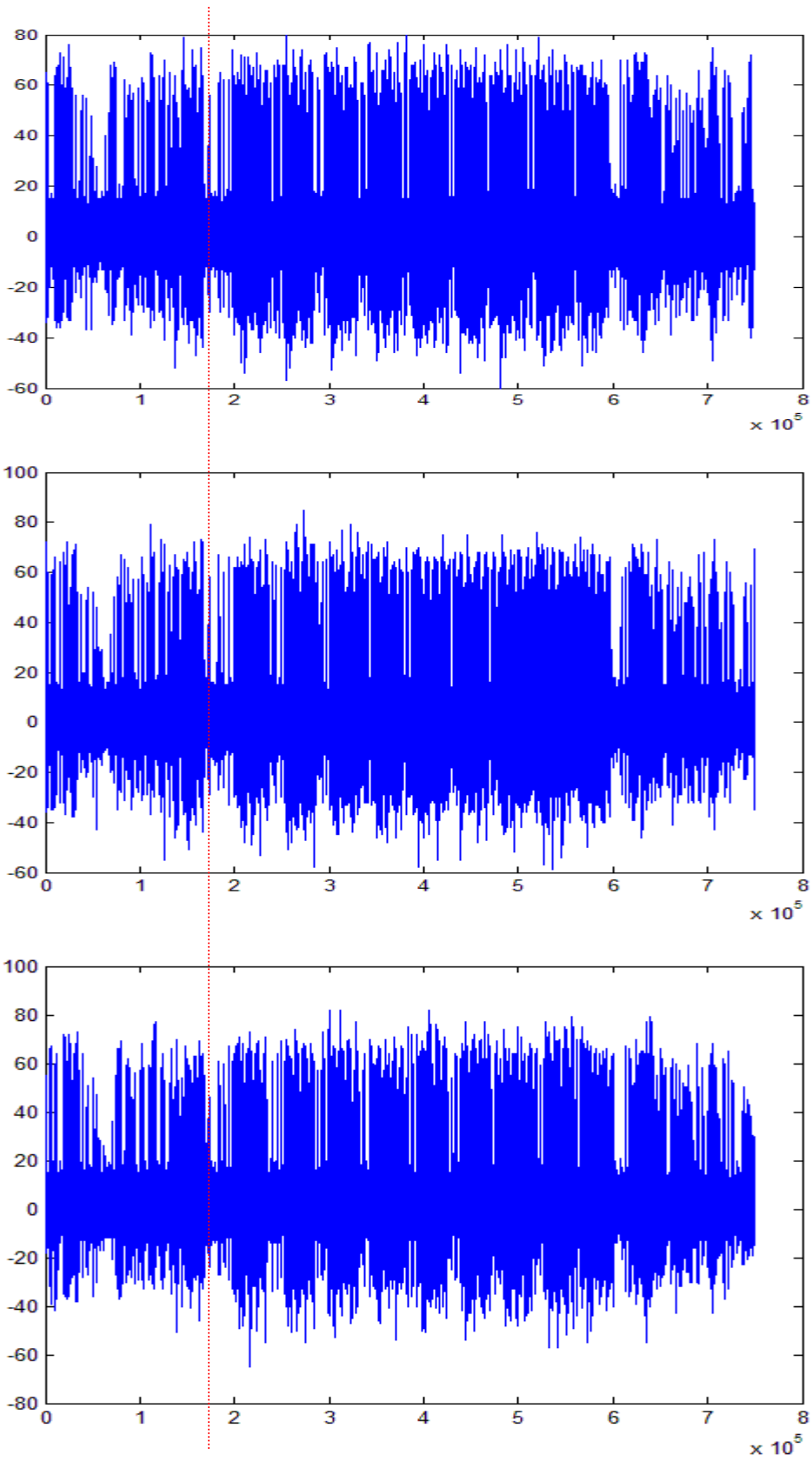
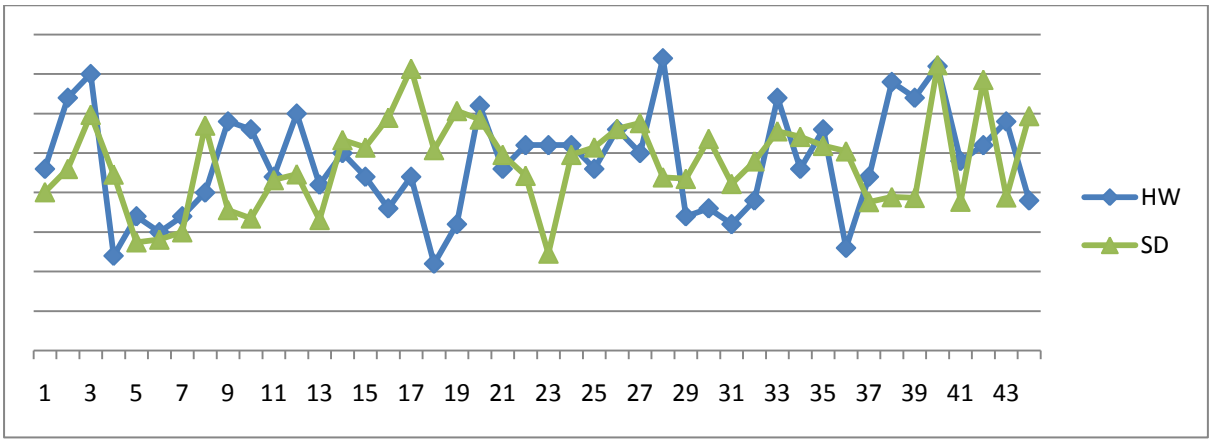


Figure 17.0 containing a sample of automatically aligned signals.

4.4- Readiness for DEMA:

Variations in signal magnitudes on average changed quite significantly throughout our recordings. Here, EM activity variance due to hamming weight alone would likely have been quite localized to the areas where 'heavier' data is being handled. As a result, average activity should be similar between recording as it is unlikely for entire data being handled throughout the encryption period to have a significantly biased overall hamming weight; enough to cause this phenomenon. Moreover, we noticed significant variations in magnitude between signals obtained by requesting the exact same AES encryption operation.

As a simple demonstration, we plot the hamming weight relating to handled plaintext throughout the first 44 requests, and the average elevation in EM activity in the first round. The hamming weight is estimated via summing the byte weights obtained from our power model for a given plaintext, while standard deviation was used as a rough estimate to overall EM activity of the first round.



Plot 18.0 illustrating relation between estimated EM activity and overall hamming weight

Here, we normalized/rescaled the data to have the same mean to allow for this quick visualization. As we notice, initially there seems to be a strong correlation between our roughly estimated EM activity and the overall hamming weight of the plaintext. This could be explained by the fact that the CPU has been running within the same speed range during the initial period where BB has been relatively inactive; thus potentially having no reason to alter CPU speeds. After a while this correlation seems to have decimated, which could be simply due to our acquisitions triggering a performance algorithm to manage ARM's speed and power consumption. However, more 'separated' trials would be needed to evaluate such an assumption.

In short, however, it seems some more effort is required before a successful DEMA can take place to minimize fluctuations in the signal which seem to 'decimate' the chances of validating a given key hypothesis. An invasive approach could be attempting to monitor or disable the interfaces/registers used to control those speeds, while non-invasive solutions could be as simple as the one described above or to attempt to normalize the

data to have fixed standard deviation values to account for changes in overall magnitude. Here however, we assume the overall activity in an entire AES execution should be relatively similar, time variations are negligible and that SD values are a good estimate to overall EM activity.

4.5- Confidence by Context:

The same event can be rationalized differently based on the context in which it appears. A smile from a parent could indicate pride or content, while from a stranger could be a sign of insanity.

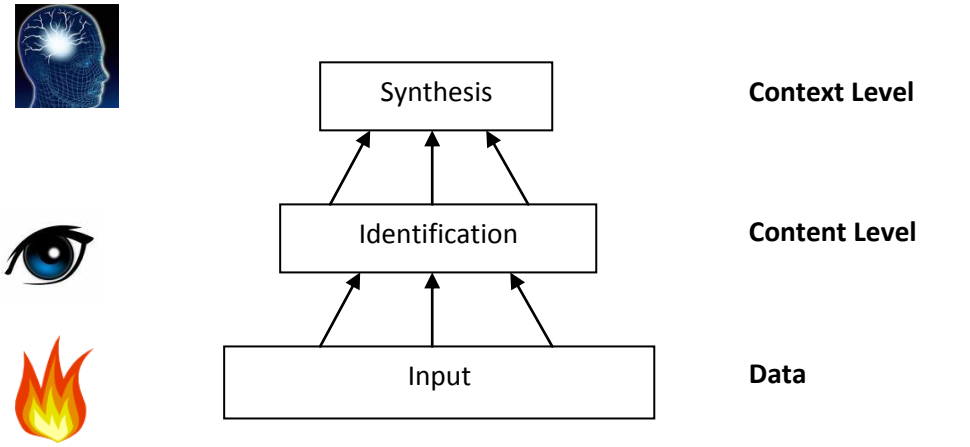


Figure 19.0 illustrating a bottom up approach in identification

Adding context to an identifiable event may thus allow for drawing conclusions with minimal amounts of data. These conclusions may be further validated and refined with the availability of more data. For example, the stranger could be a friend who simply lost a lot of weight, and was not recognized; once we hear her voice we can adjust our conclusion. With even more data, such as remembering a secret, we could have near absolute confidence in the final conclusion derived.

These are rather basic concepts, which we can apply in our scenario to identify an AES signal with limited amounts of training data and with an acceptable degree of certainty. This could be done by exposing the repetitive structure and deterministic behavior within our AES signals.

4.5.1- Data Level:

Window Selection: A window represents the portion of the vector which is to be classified. If a window is too big, we have very limited capability in making use of the repetitive structure, and adding context to the event. If widows are too small, perfect alignment would be required for each of those windows to get an adequate set of features distinguishing one window from another. We noticed from before that the first quarter of a high EM block/round is different from the remaining quarters. Thus, for our window selection, we have divided a round into two halves; where the first contained this quarter. This seemed like a reasonable trade-off for more relaxed alignment requirements (less processing).

Feature Extraction: Here there needs to be a balance in the amount of detail, where too much information could hinder the identification process, as much of it does not relate to how data is to be distinguished. In our experience, attempting to 'brute force' features from FFT transformed data resulted in training classifiers that exhibit almost 'random' classification behavior with little above 50% accuracy. Such negligible bias is not usable if we were to classify AES signals during non-repetitive executions. The process is also expensive due mainly to the FFT and complexity of the classifiers given the big number of features present in the training data.

Thus, 'hand picked' features were used, rather than utilizing generic feature extraction methods such as PCA. We have already noticed how using such features rather than relying on generic procedures resulted in enhanced accuracy and speed in the alignment process.

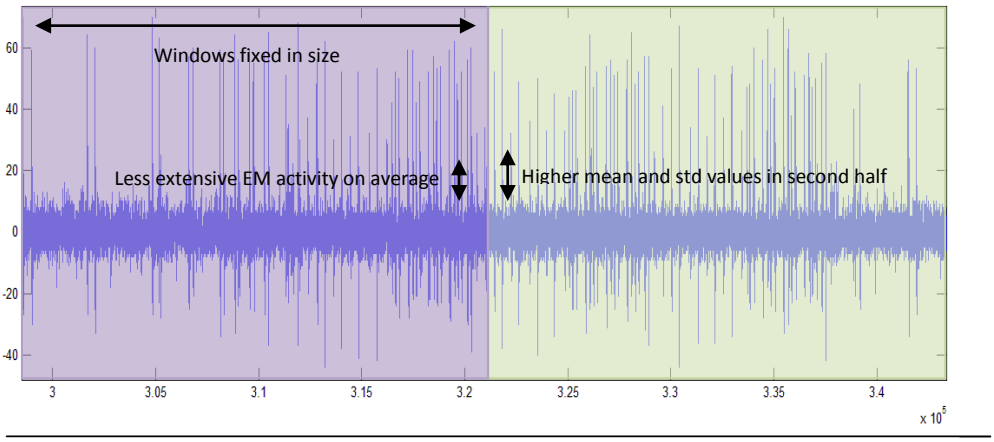


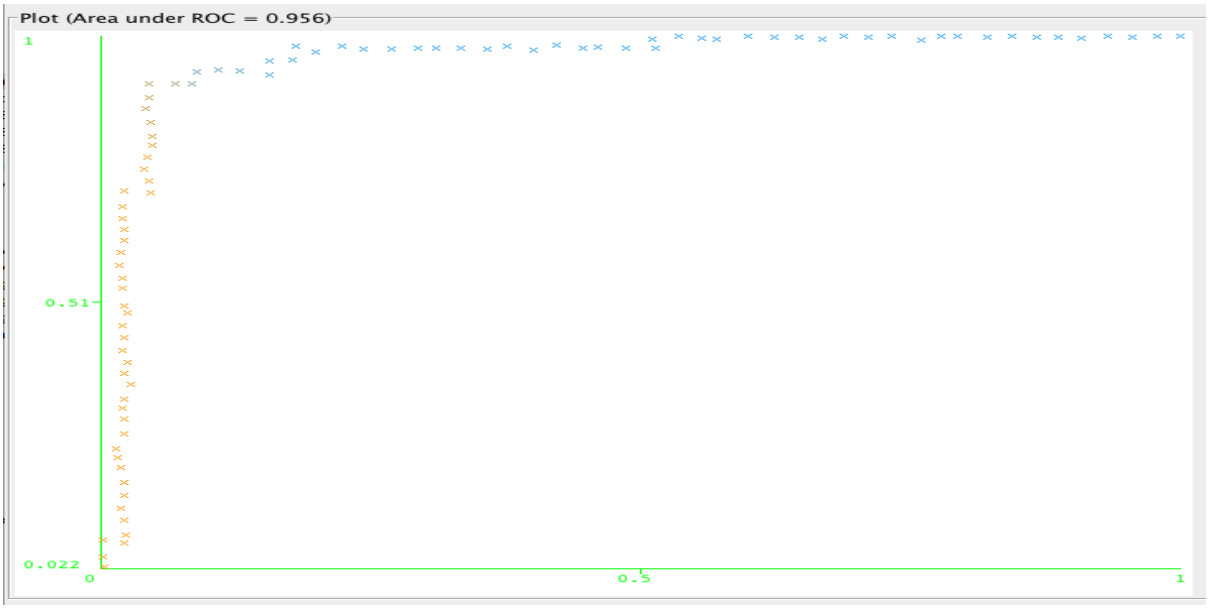
Figure 20.0 depicting the window selection process

4.5.2- Content Identification:

Different combinations of features including elements from FFT transformed vectors and classifiers such as Naive Bayes and non-optimized back propagation neural networks were used to get some initial results. It was interesting to find that Naive Bayes achieved very good results, with time indifferent features of mean and standard deviation alone. Here, it was clear to us that the feature selection process was the main key element; not choice of classifier. This also comes to show the strength of SEMA and visual inspection in depicting distinctive features, to allow for a potentially highly optimized identification process.

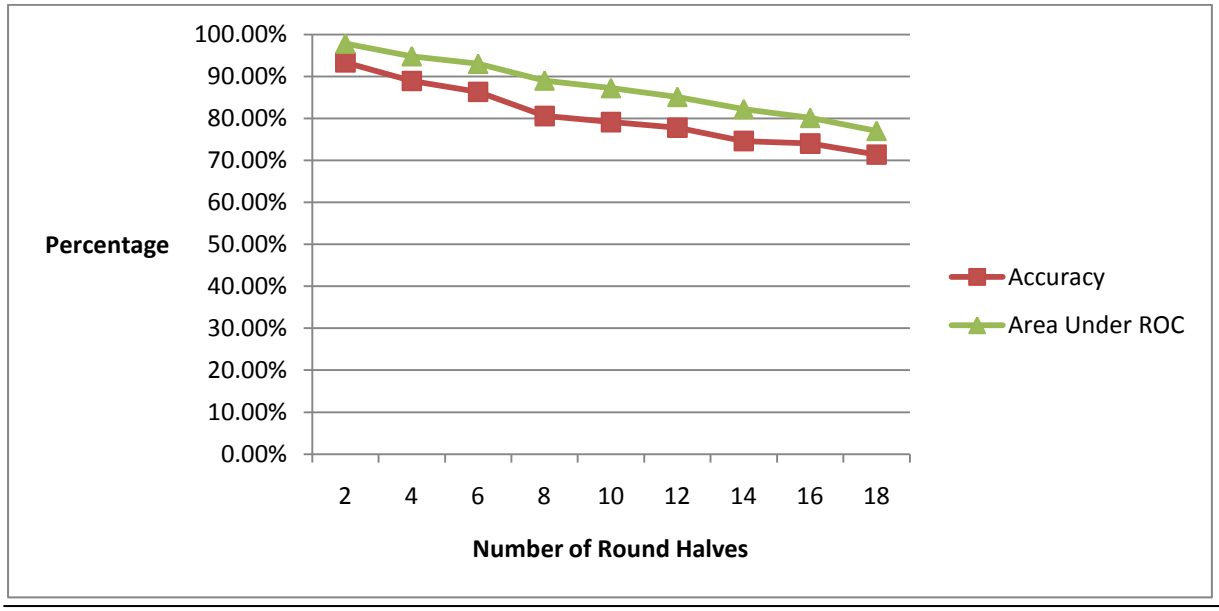
Results from identifying the two halves of the second round, on a set of 50 traces we have aligned as described before:

Accuracy using 10-cross validation: 92% *Area Under ROC Curve: 95.6%*



Plot 1.0 illustrating area under ROC curve when identifying halves in the second round

From a computational expense prospective, it is realistic to choose to align signals only once at a given point for identification from a live feed; thus, we estimate the overall accuracy to be that of the entire nine rounds, where some decrease in classification performance is expected due to the added time variation with every round.



Plot 2.0 illustrating some decline in average performance due to accumulative misalignment

Hence, an estimate to real life accuracy would be 71% and classifier's area under ROC would be 77% where we cater for potential added misalignment for every round that is identified. Nevertheless, the bias here remains rather clear in pointing out one half is of

a particular type, but results need to be combined in order to conclude that the concatenated halves likely represent an AES signal with an acceptable degree of confidence.

4.5.3- Synthesis Level:

Here we make the following definitions/assumptions:

First, we define a hypothesis H which represents our level of confidence in a signal being AES, ranging from 0 to 1.

Second, for simplicity every classification of a given half round is mapped to a fixed discrete value:

- 0.5 if classified as first half
- -0.5 if classified as second half

We then group resulting classifications in two groups:

- 1st group G1 contains results from classifying what we expect to be first halves
- 2nd group G2 contains results from classifying what we expect to be second halves

Thus, if we select N rounds <= 9 we will have N values in G1 and N values in G2.

Now, we apply a procedure similar to that used in Kocher's DPA method [2] in order to calculate a score:

Score = SUM (G1) – SUM (G2) where values in G1 are likely positive and values in G2 are likely negative.

Here, we have four main scenarios:

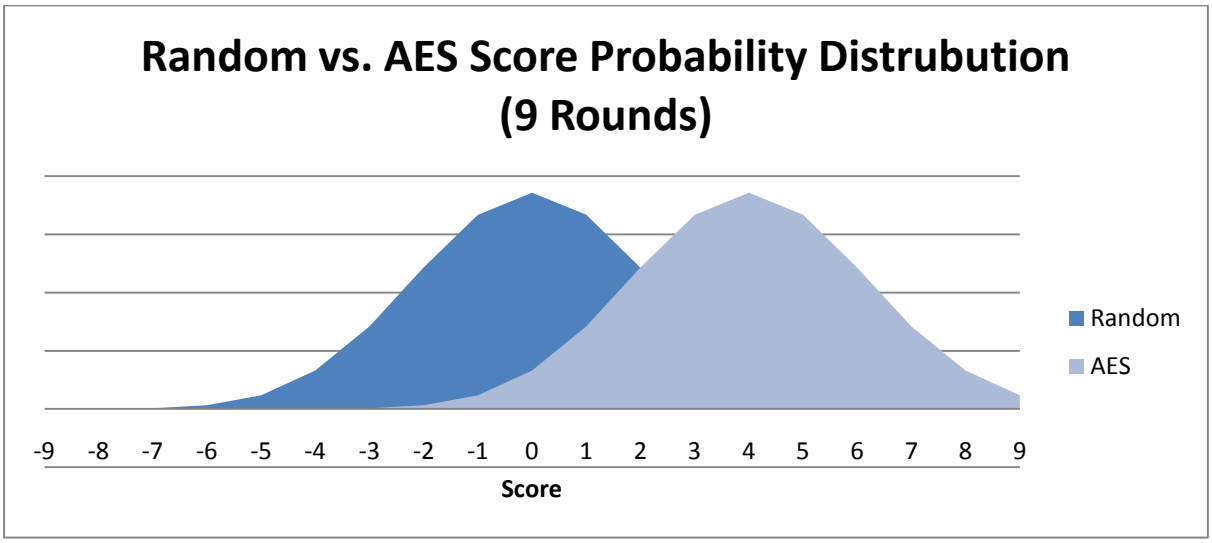
- Random Low EM activity: These will more likely be classified all as Class1, thus summation will be close to 0
- Random HIGH EM activity: These will more likely be classified all as Class2, thus summation will be close to 0
- Random High-Low activity: If N is big enough, the summation will lead to 0 [2]. In our case however, summation could possibly be either a high positive or negative value as well, if periods of low and high EM activity switch at intervals that are similar to our window size.
- AES signal: score is likely to be positive and above a certain threshold which can be found out either experimentally or statistically.

Results from 50 signals using last 9 rounds (18 halves):

Average score: *3.89*

Standard Deviation: *0.79*

However, since only 50 signals were used (450 halves), it would be more realistic to assume AES signal scores have normal distribution with a mean of 3.89. We also assume the worst case scenario where signal classification for non-AES signals randomly changes at the same window interval we have chosen for classification.



Plot 3.0 illustrating simulated probability distribution for nine rounds opposed to random data

Here we notice that an estimated confidence level of a signal being AES at a score of 2 is little below half (around 47%) where a signal is about as likely be random as an AES signal. The linear Hypothesis value H could be calculated as follows to represent those odds:

H = Confidence(AES|Score) , where H ranges from 0 to 1.

4.5.4- Simple Generalization (by Abstraction):

We noticethat the synthesis 'node' outputs a hypothesis of 0 to 1, allowing it to act as an identification 'node' if required. Here, we could have a higher level of perception built on several synthesis nodes as well as identification nodes. However, it is logical to have one top synthesis node which ultimately draws the required conclusion "Sending e-mail?". Other nodes here will be either:

- Leaf nodes: which are purely identification nodes as seen before, and actually process our extracted features
- Intermediate nodes: these act as synthesis nodes for lower level nodes, and as identification nodes for those above. They do not process raw data, but rather pre-identified data from lower levels

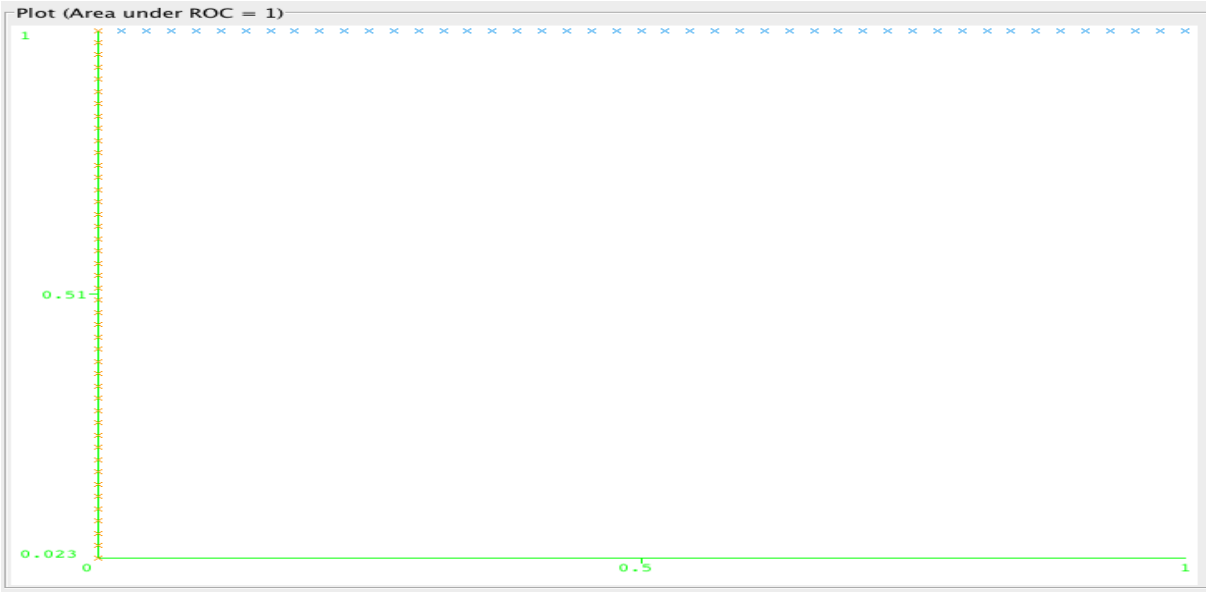
Example:

Assume we now want to include identification results from the first round but do not want to re-write our existing code.

Classification results using Naive Bayes, Neural Networks and others:

Accuracy: 100%

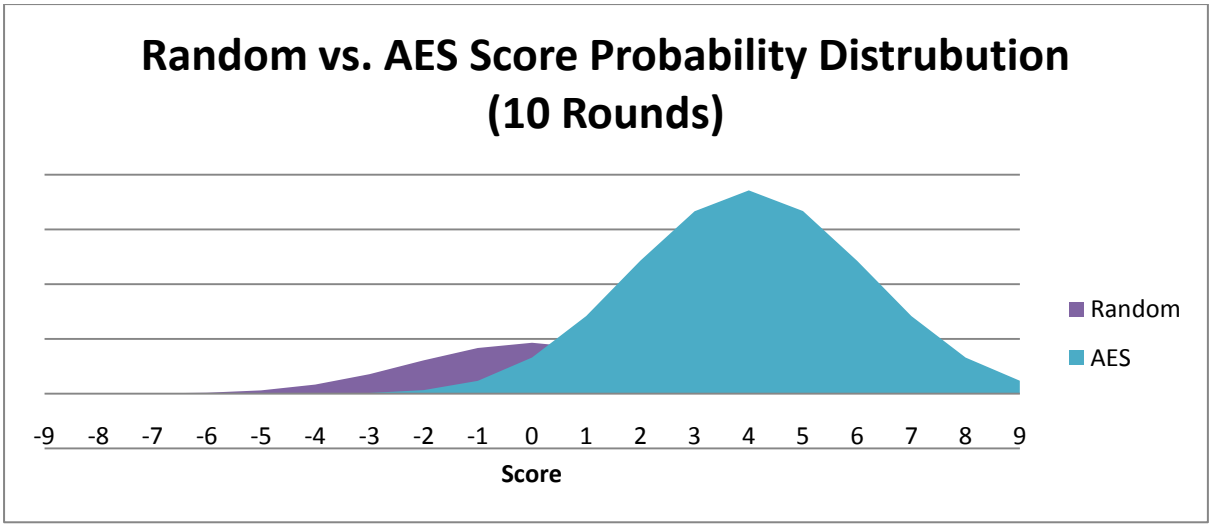
Area Under ROC: 100%



Plot 4.0 illustrating ROC curve for identifying the first round

This had been somewhat anticipated as we have aligned our signals after the first round immediately, and due to the deterministic low EM activity -table loading?- before the first round. Improved results are expected with other rounds if all were aligned in this matter and not using one reference point.

Here, we can construct a context (synthesis) node which utilizes the new identification node –acting as a leaf node- and the existing context node – acting as an intermediate node-. The method for combining those results can be different than before where we can utilize the fact that we have a 'perfect' ROC curve and use identification results from the first round as a filter instead. As such it is likely that only ¼ of random signals will pass the test if both halves of the first round are classified correctly.



Plot 5.0 illustrating simulated probability distribution for nine rounds opposed to random data

Thus, our 'confidence' levels with the same score 2 increases to around 70% arguably by being able to fit our evaluation into a 'grander' context, which allows us to set a reasonable threshold where any positive score >=1 is accepted for further analysis. One could image such a 'tree' potentially growing further to accommodate more complex patterns as well. The disadvantage of such a construct however, is that can be closely tied to a given implementation/device, and although might provide an advantage in that inner working are rather transparent –as opposed to black box alternatives-, can be costly to build and potentially inflexible to change if levels of abstraction are poorly designed

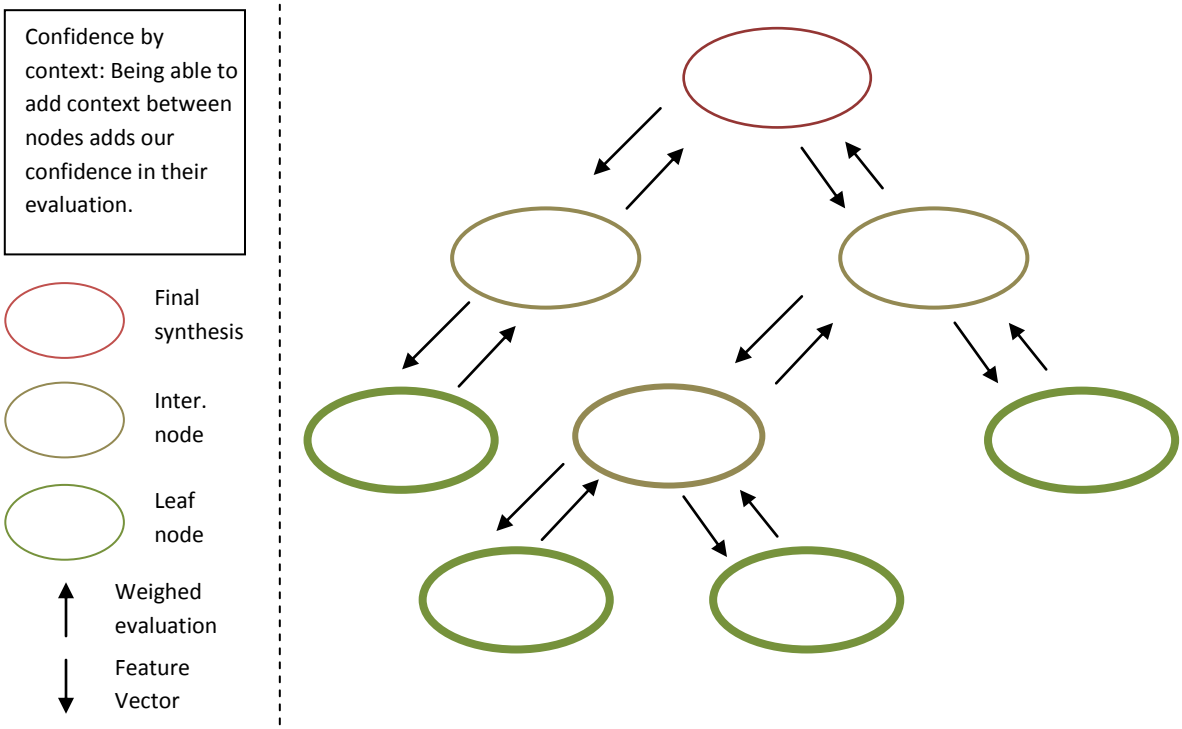


Figure 21.0 illustrating information flow in a confidence by context tree

Chapter 5: Future Work

Although this work aims to facilitate for future SCA, there remains much work that need be done before a realistic attack can take place. The first sensible step could be to devise a method to efficiently obtain signals which do not exhibit the fluctuations encountered. This could be followed by testing for successful correlation between the signals and hamming weight of our plaintext. Once complete, we can then utilize DEMA to validate our assumptions regarding implementation details of BB's AES code, to extract secret key information and to find 'interesting points' to build potential templates that can potentially obtain the key in a very limited number of traces. Such a requirement seems necessary if we were to mount a realistic attack.

Here, one could attempt to build templates which do not rely on the plaintext at all. As we have seen there is an added complexity of having to know what is being encrypted at a given point of time to allow launching a known plaintext SCA, especially in a CBC encryption mode context. Here, we could target/model the signal and noise associated with key scheduling operations in particular. In this case, even if a naïve approach is used to build 2^{32} templates associated with every possible complete key row scheduling in an optimized method, such a huge storage space is barely incomprehensible even in today's standards.

Now remain four main problems: The first is that a real life attack must accommodate fluctuations in speed. Here an approach where for every possible key we create N templates where N equals number of speeds could be both flawed and expensive. Flawed, as speeds can change within a single operation [56] as well as multiple operations. This means that even the simplest of templates that model a single or a very small number of operations such as that relating to an S-Box lookup -on a XORed key and plaintext- could fail, even if we considered all speeds. Such an approach is potentially over expensive as well, as BB may have simply used a small subset of the speed range provided. Here, it would be sensible to use an iterative approach were we attempt to 'capture' or model all possible variations rather than blindly create templates for all fixed speeds. As despite how complex performance management algorithms could be[55], there must be finite sets of speed fluctuation 'profiles' that could take place for a given set of operations. Here, a key could have multiple templates –one per fluctuation profile-, as noise associated with a potentially huge number of fluctuation profiles might possibly no longer be an accurate overall approximation to expected noise associated with a given key, as noise in such a complex setup may no longer be confined to simple Gaussian distribution.

Another problem is that a template could require aligning interesting points requires and time domain consistency if were to be as efficient and effective as possible. Such inflexibility might hinder our chances of success in a realistic environment or significantly add to the 'cost' of the attack. Thus, alternatively we can use templates

which operate in the frequency domain instead [57]. To estimate interesting points here, procedures such as those described in [57] to elect points which represent the most variation amongst noise whether PCA or a less expensive method where points which have maximal differences between runs on the same data, followed by depicting sufficiently spaced points can be used. Here, such a method is a potential candidate solution for added flexibility.

The third problem is handling BB's newly introduced mitigation measurements against SCA [58]. These include masking, table splitting and adding dummy operations. Those mentioned however may have potentially already been catered for in our approach, where templates have been proven potentially just as effective in masked and unmasked implementations [59], due to the fact that we are attacking key scheduling operations and as adding dummy operations is likely ineffective in a -time domain indifferent- frequency based template attack. Thus, although a separate template construction process might be required, such BB mitigations could be potentially 'futile'.

Some remaining 'logistic' problems are that the noise during an attack needs to be consistent with the 'trained' templates as well. This could be achieved by a potentially advanced digital signal processing, or by creating an environment, where user will likely position the device in a favorable location. For example we could fit trains with BB docks/chargers of fixed proximity to our EM probe(s). For EM recording, special oscilloscope memory space requirements would be needed to support recording live feeds for significant periods of time upon a given EM threshold and/or BB docking. A streamlined process of initial alignment and identification can take place to minimize disk storage requirements by discarding non-AES signals. However, again more effort needs to be done to modify our automated alignment and identification processes to cater for noise imposed by having the BB in its original casing. In such conditions however, it can be argued that although such a deterministic feature as the one used to align our signals is no longer realistic to seek, us using simpler variations such as mean and standard deviation for identification, although could be scaled down heavily, could be detectable still when moving away from the CPU, allowing for a simple identification procedure such as the one devised to remain effective if we allow our window to slide and allow for any positive or negative identification bias to be accepted, especially if only a few traces are needed where no plaintext knowledge is required.

Bibliography:

1. S. Mangard, E. Oswald and T. Popp, *Power Analysis Attacks - Revealing the Secret of Smart Cards*. 2007, NY: Springer Science + Business Media.
2. P. Kocher et.al. *Differential Power Analysis*, in *Advances in Cryptology-Proceedings of Crypto*. 1999, Lectures in Computer Science.
3. D. Agrawal et.al, *The EM Side-Channel(s).* Lecture notes in Computer Science, 2003.
4. C. Gebotys et.al, *EM Analysis of Rijndael and ECC on a Wireless Java-Based PDA.* Lecture Notes in Computer Science, 2005. **3659**.
5. O. Aciiçmez, W. Schindler, and Ç. Koç, *Cache Based Remote Timing Attack on the AES*. 2006. p. 271-286.
6. P. Kocher, *Timing Attacks on Implementation of Diffie-Hellman, RSA, DSS and other Systems*, in *16th Annual International Cryptology Conference*. 1996: Santa Barbra, California, USA.
7. D. Osvik, A. Shamir, and E. Tromer, *Cache Attacks and Countermeasures: The Case of AES*. 2006. p. 1-20.
8. J. Daemen and V. Rijmen, *The design of Rijndael: AES the advanced encryption standard.* 2002, Springer-Verlag.
9. J. Bonneau and I. Mironov, *Cache-Collision Timing Attacks Against AES*, in *Cryptographic Hardware and Embedded Systems - CHES 2006*. 2006, Springer Berlin / Heidelberg. p. 201-215.
10. M. Neve and J. P. Seifert, *Advances on Access-Driven Cache Attacks on AES*. 2007, Springer Berlin / Heidelberg. p. 147-162.
11. G. Bertoni et al. *AES Power Attack Based on Induced Cache Miss and Countermeasure*. in *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*. 2005.
12. T. Mitchel, *Machine Learning*. 1997: McGraw Hill.
13. S. Ho *EM Analysis of ECC Computations on Mobile Devices*. 2005; Available from: http://optimal.vlsi.uwaterloo.ca/NEW/Simon_thesis.pdf.
14. RIM. *About RIM*. 2009; Available from: http://www.rim.com/company/index.shtml.
15. NEWS, C. *RIM co-founder donates $50M to Waterloo physics centre*. 2008; Available from: http://www.cbc.ca/technology/story/2008/06/05/lazaridis-perimeter.html.
16. Waterloo, U.o. *RIM founder named chancellor*. 2009; Available from: http://newsrelease.uwaterloo.ca/news.php?id=2764.
17. GSMWORLD. *GSM World Coverage*. 2009; Available from: http://www.gsmworld.com/roaming/GSM_WorldPoster2009A.pdf.
18. BlackBerry. *Wireless Data Security Features*. 2009; Available from: http://uk.blackberry.com/ataglance/security/features.jsp.
19. Oracle. *Siebel CRM Mobile for BlackBerry Whitepaper*. 2008; Available from: http://uk.blackberry.com/campaign/mobilecrm/siebel_crm_mobile_bb_white_paper.pdf.
20. Blackberry. *Stored Data Security Features*. 2009; Available from: http://uk.blackberry.com/ataglance/security/features.jsp#tab_tab_stored_data.
21. BlackBerry. *Java Application Development*. 2009; Available from: http://na.blackberry.com/eng/developers/javaappdev/.
22. BlackBerry. *Application Development Advanced Features*. 2009; Available from: http://na.blackberry.com/eng/developers/javaappdev/advancedfeatures.jsp.
23. Blackberry. *Blackberry IT Policy at a Glance*. 2009; Available from: http://uk.blackberry.com/ataglance/security/it_policy.jsp.
24. BlackBerry. *Blackberry Complete Policy Reference Guide*. 2009; Available from: http://docs.blackberry.com/eng/deliverables//3801/Policy_Reference_Guide.pdf.

25. BlackBerry. *Blackberry Device Activation Technical Whitepaper*. 2009; Available from: http://docs.blackberry.com/en/admin/deliverables/3643/BlackBerry_Wireless_Enterprise_Activation_Technical_White_Paper.pdf.
26. Certicom. *MQV: Efficient and Authenticated Key Agreement*. 2009; Available from: http://www.certicom.com/index.php/mqv.
27. BlackBerry. *BlackBerry Enterprise Solution Security Release 4.1 Technical Overview*. 2006; Available from: http://na.blackberry.com/eng/ataglance/get_the_facts/xHCO-BlackBerry_Enterprise_Solution_Security_version_4.pdf.
28. M. Sharifi et al., *A Zero Knowledge Password Proof Mutual Authentication Technique Against Real-Time Phishing Attacks*. 2007. p. 254-258.
29. H. Krawczyk, *HMQV: A High-Performance Secure Diffie-Hellman Protocol*. 2005. p. 546-566.
30. Ericsson. *Ericsson MSC Server Blade Cluster*. 2008; Available from: http://www.ericsson.com/ericsson/corpinfo/publications/review/2008_03/files/Blade.pdf.
31. Alcatel-Lucent. *Alcatel-Lucent CDMA Mobile Switching Center (MSC)*. 2009; Available from: http://www.alcatel-lucent.com/wps/portal/products/detail?LMSG_CABINET=Solution_Product_Catalog&LMSG_CONTENT_FILE=Products/Product_Detail_000004.xml#tabAnchor1.
32. Nortel. *Nortel Gateway GPRS Support Node*. 2009; Available from: http://www2.nortel.com/go/product_content.jsp?segId=0&parId=0&prod_id=50143.
33. Cisco. *SAMI-based GGSN, 7613*. 2008; Available from: http://www.cisco.com/en/US/prod/collateral/wireless/wirelssw/ps873/CA-Iometrix-Cisco-GGSN7613D.pdf.
34. BlackBerry. *Blackberry Approvals and Certifications*. 2009; Available from: http://na.blackberry.com/eng/ataglance/security/certifications.jsp.
35. BlackBerry. *Software Download for AT&T*. 2009; Available from: https://www.blackberry.com/Downloads/entry.do?code=577BCC914F9E55D5E4E4F82F9F00E7D4.
36. SUN-MicroSystems. *The K virtual machine (KVM)*. 2009; Available from: http://java.sun.com/products/cldc/wp/.
37. SUN-MicroSystems. *Connected Limited Device Configuration HotSpot™ Implementation*. 2004; Available from: http://java.sun.com/j2me/docs/pdf/CLDC_HI112.pdf.
38. SUN-MicroSystems. *Mobile Information Device Profile (MIDP); JSR 37, JSR 118 Overview*. 2009; Available from: http://java.sun.com/products/midp/overview.html#2.
39. BlackBerry. *BlackBerry MIDlet Developer Guide*. 2006; Available from: http://docs.blackberry.com/en/developers/deliverables/612/BlackBerry_MIDlet_Developer_Guide.pdf.
40. BlackBerry. *Application Signing Keys*. 2009; Available from: http://na.blackberry.com/eng/developers/javaappdev/codekeys.jsp.
41. RIM. *Blackberry™ Cryptographic Kernel Version 3.3*. 2003; Available from: http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp312.pdf.
42. Sam Mason, E.K. *Native and Java ME Development on Symbian OS*. 2008; Available from: http://developer.symbian.com/main/downloads/papers/Native_And_Java_ME_Dev_On_SymbianOS_%20v1.1.pdf.
43. Android. *What is Android?* 2009; Available from: http://developer.android.com/guide/basics/what-is-android.html.
44. ARM. *Jazelle Technology Overview*. 2009; Available from: http://www.arm.com/products/multimedia/java/jazelle.html.
45. SUN-MicroSystems. *Class Midlet Javadoc Entry*. 2006; Available from: http://java.sun.com/javame/reference/apis/jsr118/javax/microedition/midlet/MIDlet.html.

46. BlackBerry. *BlackBerry Java Development Environment: Fundamentals Guide*. 2008; Available from: http://docs.blackberry.com/en/developers/deliverables/4526/BlackBerry_Java_Development_Environment-4.7.0-US.pdf.
47. BlackBerry. *Blackberry CLDC Overview*. 2009; Available from: http://docs.blackberry.com/en/developers/deliverables/5827/CLDC_applications_446988_11.jsp.
48. BlackBerry. *BlackBerry Java Development Environment*. 2009; Available from: http://na.blackberry.com/eng/developers/javaappdev/javadevenv.jsp.
49. BlackBerry. *BlackBerry JDE Plug-in for Eclipse*. 2009; Available from: http://na.blackberry.com/eng/developers/javaappdev/javaeclipseplug.jsp.
50. SUN-MicroSystems. *Programming the BlackBerry With J2ME*. 2009; Available from: http://developers.sun.com/mobility/midp/articles/blackberrydev/.
51. BlackBerry. *Package net.rim.device.api.crypto Javadoc Entry*. 2009; Available from: http://www.blackberry.com/developers/docs/4.0.2api/net/rim/device/api/crypto/package-summary.html.
52. BlackBerry, *BlackBerry Signing Authority Tool*. 2005.
53. BlackBerry. *Protecting the BlackBerry device platform against malware*. 2006; Available from: http://www.blackberry.com/solutions/resources/Protecting_the_BlackBerry_device_platform_against_malware.pdf.
54. BlackBerry. *Stay Connected with your life with Facebook for Blackberry*. 2009; Available from: http://na.blackberry.com/eng/devices/features/social/facebook.jsp?
55. M. Weiser et al., *Scheduling for Reduced CPU Energy*. 1996, The Kluwer International Series in Engineering and Computer Science. p. 449-471.
56. ARM. *Intelligent Energy Controller - Technichal Overview*. 2003-2005; Available from: http://infocenter.arm.com/help/topic/com.arm.doc.dto0005c/DTO0005.pdf.
57. C. Rechberger and E. Oswald, *Practical Template Attacks*. 2005, Springer Berlin / Heidelberg. p. 440-456.
58. BlackBerry, *How the BlackBerry Enterprise Solution uses an AES encryption algorithm* 2009.
59. E. Oswald and S. Mangard, *Template Attacks on Masking—Resistance Is Futile*. 2006, Springer Berlin / Heidelberg. p. 243-256.

Appendix A: Code & BB Server User Documentation

Notice: The following code segments may contain portions of code provided by Research In Motion, and other public resources. Only that has been modified to fit our needs has been included.

```
/*
 * AesExecuter.java
 *
 * © Research In Motion Limited, 2003-2003
 * Confidential and proprietary.
 */

package edu.bristol.crypto.sca.rim.run;

import java.io.*;

import edu.bristol.crypto.sca.rim.data.Hex;
import edu.bristol.crypto.sca.rim.data.Parameter;

import net.rim.device.api.crypto.*;
import net.rim.device.api.system.Backlight;
import net.rim.device.api.system.LED;

// TODO: Auto-generated Javadoc
/**
 * The Class AESEncryptor.
 */
public class AESEncryptor implements Executer{

   /** The key. */
   private AESKey key;

   /** The message. */
   private byte [] message;

   /** The trigger. */
   private String trigger;

   /** The wait after trigger. */
   private int waitAfterTrigger;

   /** The wait after encryption. */
   private int waitAfterEncryption;

      /** The output stream. */
      private ByteArrayOutputStream outputStream;

   /** The encryption engine. */
   private AESEncryptorEngine encryptionEngine;

   /** The formatter engine. */
   private PKCS5FormatterEngine formatterEngine;

   /** The encryptor. */
   private BlockEncryptor encryptor;

   /**
    * Instantiates a new aES encryptor.
```

```
 */
public AESEncryptor() {
   key = new AESKey();
   waitAfterEncryption=waitAfterTrigger=0;
   }

/* (non-Javadoc)
 * @see edu.bristol.crypto.sca.rim.run.Executer#run()
 */
public String run(){
   try {
            encryptionEngine = new AESEncryptorEngine( key );
        } catch (CryptoTokenException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (CryptoUnsupportedOperationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (Exception ex){

      }
   outputStream=new ByteArrayOutputStream();
   formatterEngine = new PKCS5FormatterEngine( encryptionEngine );
   encryptor = new BlockEncryptor( formatterEngine, outputStream );

   if (waitAfterTrigger==0&&waitAfterEncryption==0){
        System.out.println("NO WAIT");
        executeNoWait();
   }
   else if(waitAfterTrigger!=0&&waitAfterEncryption!=0){
        System.out.println("WAIT BEFORE AND AFTER");
        executePrePostWait();
   }
   else if(waitAfterTrigger!=0){
        System.out.println("WAIT BEFORE ONLY");
        executePreWait();
   }
   else {
        System.out.println("WAIT AFTER ONLY");
        executePostWait();
   }
      System.out.println("Hi14");

   return Hex.toHex(outputStream.toByteArray());
}

/**
 * Execute no wait.
 */
private void executeNoWait(){
   try {
        LED.setState(LED.STATE_ON);
            encryptor.write( message );
            LED.setState(LED.STATE_OFF);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (Exception ex){
```

```
        }

}

/**
 * Execute pre wait.
 */
private void executePreWait(){
   try {
        LED.setState(LED.STATE_ON);
      Thread.sleep(waitAfterTrigger);
        encryptor.write( message );
        LED.setState(LED.STATE_OFF);
   } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
   } catch (IOException e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
                 } catch (Exception ex){

     }


}

/**
 * Execute post wait.
 */
private void executePostWait(){
   try {
        LED.setState(LED.STATE_ON);
        encryptor.write( message );
      Thread.sleep(waitAfterEncryption);
        LED.setState(LED.STATE_OFF);
   } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
   } catch (IOException e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
                 } catch (Exception ex){

     }

}

/**
 * Execute pre post wait.
 */
private void executePrePostWait(){

   try {
        LED.setState(LED.STATE_ON);
      Thread.sleep(waitAfterTrigger);
      encryptor.write( message );
      Thread.sleep(waitAfterEncryption);
        LED.setState(LED.STATE_OFF);
   } catch (IOException e) {
```

```
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } catch (Exception ex){

        }
}

    /**
     * Sets the aes key.
     *
     * @param aesKey the new aes key
     */
    public void setAesKey(AESKey aesKey) {
        this.key = aesKey;
    }

/**
 * Gets the aes key.
 *
 * @return the aes key
 */
public byte[] getAesKey() {
        try {
            return key.getData();
        } catch (CryptoTokenException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (CryptoUnsupportedOperationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (Exception ex){

        }
        //Should not happen
        return null;
    }


    /**
     * Sets the aes key.
     *
     * @param aesKey the new aes key
     */
    public void setAesKey(byte[] aesKey) {
        this.key = new AESKey(aesKey);
    }

    /**
     * Gets the output stream.
     *
     * @return the output stream
     */
    public ByteArrayOutputStream getOutputStream() {
        return outputStream;
    }
```

```
/**
 * Sets the output stream.
 *
 * @param outputStream the new output stream
 */
public void setOutputStream(ByteArrayOutputStream outputStream) {
	this.outputStream = outputStream;
}

/**
 * Gets the encryption engine.
 *
 * @return the encryption engine
 */
public AESEncryptorEngine getEncryptionEngine() {
	return encryptionEngine;
}

/**
 * Sets the encryption engine.
 *
 * @param encryptionEngine the new encryption engine
 */
public void setEncryptionEngine(AESEncryptorEngine encryptionEngine) {
	this.encryptionEngine = encryptionEngine;
}

/**
 * Gets the formatter engine.
 *
 * @return the formatter engine
 */
public PKCS5FormatterEngine getFormatterEngine() {
	return formatterEngine;
}

/**
 * Sets the formatter engine.
 *
 * @param formatterEngine the new formatter engine
 */
public void setFormatterEngine(PKCS5FormatterEngine formatterEngine) {
	this.formatterEngine = formatterEngine;
}

/**
 * Gets the encryptor.
 *
 * @return the encryptor
 */
public BlockEncryptor getEncryptor() {
	return encryptor;
}

/**
 * Sets the encryptor.
 *
 * @param encryptor the new encryptor
 */
```

```
    public void setEncryptor(BlockEncryptor encryptor) {
        this.encryptor = encryptor;
    }

/**
 * Gets the key.
 *
 * @return the key
 */
public AESKey getKey() {
        return key;
    }

    /**
     * Sets the key.
     *
     * @param key the new key
     */
    public void setKey(AESKey key) {
        this.key = key;
    }

    /**
     * Gets the message.
     *
     * @return the message
     */
    public byte[] getMessage() {
        return message;
    }

    /**
     * Sets the message.
     *
     * @param message the new message
     */
    public void setMessage(byte[] message) {
        this.message = message;
    }

    /**
     * Gets the trigger.
     *
     * @return the trigger
     */
    public String getTrigger() {
        return trigger;
    }

    /**
     * Sets the trigger.
     *
     * @param trigger the new trigger
     */
    public void setTrigger(String trigger) {
        this.trigger = trigger;
    }

    /**
```

```
     * Gets the wait after trigger.
     *
     * @return the wait after trigger
     */
    public int getWaitAfterTrigger() {
        return waitAfterTrigger;
    }

    /**
     * Sets the wait after trigger.
     *
     * @param waitAfterTrigger the new wait after trigger
     */
    public void setWaitAfterTrigger(int waitAfterTrigger) {
        this.waitAfterTrigger = waitAfterTrigger;
    }

    /**
     * Gets the wait after encryption.
     *
     * @return the wait after encryption
     */
    public int getWaitAfterEncryption() {
        return waitAfterEncryption;
    }

    /**
     * Sets the wait after encryption.
     *
     * @param waitAfterEncryption the new wait after encryption
     */
    public void setWaitAfterEncryption(int waitAfterEncryption) {
        this.waitAfterEncryption = waitAfterEncryption;
    }

    /* (non-Javadoc)
     * @see
edu.bristol.crypto.sca.rim.run.Executer#setParameter(edu.bristol.crypto.sca.rim.data.Parameter)
     */
    public void setParameter(Parameter parameter){
        if(parameter.getKey()!=null){
            setAesKey(parameter.getKey());
        }
        setMessage(parameter.getPlainText());
        setTrigger(parameter.getTrigger());
        setWaitAfterTrigger(parameter.getWaitAfterTrigger());
        setWaitAfterEncryption(parameter.getWaitAfterEncryption());
    }
}
```

```
/*
 * UsbServer.java
 *
 */

package edu.bristol.crypto.sca.rim.server;

import java.io.*;
import java.util.*;
import javax.microedition.io.*;


import edu.bristol.crypto.sca.rim.data.Parameter;
import edu.bristol.crypto.sca.rim.postoffice.Parser;
import edu.bristol.crypto.sca.rim.postoffice.PostOffice;

import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;

// TODO: Auto-generated Javadoc
/**
 * The Class UsbServer.
 */
public final class UsbServer extends UiApplication implements KeyListener, TrackwheelListener/*
extends DemoApp*/ {

   //constants
   /** The Constant CHANNEL. */
   private static final String CHANNEL = "R";

   //members
   /** The _output. */
   private RichTextField _output;

   /** The _usb thread. */
   private UsbThread _usbThread;
   //private DESKey desKey;
   //private AESKey aesKey;
   /** The post office. */
   private PostOffice postOffice;

   /** The _connect gcf. */
   private MenuItem _connectGCF = new MenuItem("Start Server", 10, 10) {
      public void run()
      {
            postOffice=new PostOffice();
            message("Server Started..");
            onExit(); //cleanup the old thread if present
            _usbThread = new GCFUsbThread();
            connect();
      }
   };

   /** The _close item. */
   private MenuItem _closeItem = new MenuItem("Exit", 200000, 10) {
      public void run()
```

```
    {
      onExit();
      System.exit(0);
    }
  };

  //statics
/*   private static ResourceBundle _resources =
ResourceBundle.getBundle(UsbDemoResource.BUNDLE_ID,
UsbDemoResource.BUNDLE_NAME);
*/
  /**
   * The main method.
   *
   * @param args the arguments
   */
  static public final void main(String[] args)
  {
    new UsbServer().enterEventDispatcher();

  }

  //inner classes

  /**
   * The Class UsbThread.
   */
  private abstract class UsbThread extends Thread {

    /* (non-Javadoc)
     * @see java.lang.Thread#run()
     */
    public void run()
    {
      try {
        init();
        Parser parser=new Parser();
        String s = read(";");
        message("received: " + s);
        Parameter paramter=parser.parse(s);

        if(paramter.isResponseFlag()){
          String response=postOffice.execute(paramter);

          if (response==null){
            response="ERROR;";
          }
          message("Sending: \n"+response);
          write(response);

        }
        else{//asynchronous
            close();
            onExit();
          _usbThread = new GCFUsbThread();
          connect();
          postOffice.execute(paramter);
          return;
        }
```

```
            close();

          } catch (IOException e) {
          //     message(e.toString());
          } catch (Exception ex){
//              message(ex.toString());
          }

            onExit();
            _usbThread = new GCFUsbThread();
            connect();

//          }
      }

      /**
       * Inits the.
       *
       * @throws IOException Signals that an I/O exception has occurred.
       */
      abstract protected void init() throws IOException;

      /**
       * Write.
       *
       * @param s the s
       *
       * @throws IOException Signals that an I/O exception has occurred.
       */
      abstract protected void write(String s) throws IOException;

      /**
       * Read.
       *
       * @param match the match
       *
       * @return the string
       *
       * @throws IOException Signals that an I/O exception has occurred.
       */
      abstract protected String read(String match) throws IOException;

      /**
       * Close.
       */
      abstract public void close();
    }

    /**
     * The Class GCFUsbThread.
     */
    private final class GCFUsbThread extends UsbThread
    {

      /** The _dis. */
      private DataInputStream _dis;

      /** The _dos. */
```

```
    private DataOutputStream _dos;

    /** The _sc. */
    private StreamConnection _sc;

    /* (non-Javadoc)
     * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#init()
     */
    protected void init() throws IOException
    {
      //Using the connector interface
      _sc = (StreamConnection)Connector.open("comm:USB;channel=" +
CHANNEL);

      _dis = _sc.openDataInputStream();
      _dos = _sc.openDataOutputStream();
    }

    /* (non-Javadoc)
     * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#write(java.lang.String)
     */
    protected void write(String s) throws IOException
    {
      _dos.writeUTF(s);
      _dos.flush();
    }

    /* (non-Javadoc)
     * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#read(java.lang.String)
     */
    protected String read(String terminator) throws IOException
    {
      //int len = match.length();
      byte[] data = new byte[1];
      boolean flag=false;
      String s=null;
      while(!flag){
        _dis.read(data, 0, 1);
        String tempString = new String(data);
        if(tempString.equals(terminator)){
          flag=true;
        }
        if(s==null){
          s=tempString;
        }
        else{
          s+=tempString;
        }
      }
      return  s;
    }

    /* (non-Javadoc)
     * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#close()
     */
    public void close()
    {
      try {
        /*parent.message("closing connections...");*/
```

```
            if ( _dis != null ) _dis.close();
            if ( _dos != null ) _dos.close();
            if ( _sc != null ) _sc.close();
            /*parent.message("connections closed");*/
          } catch (IOException e) {
            /*parent.message(e.toString());*/
          }
        }
      }

      /**
       * The Class LowLevelUsbThread.
       */
      private final class LowLevelUsbThread extends UsbThread implements
    USBPortListener
      {

        /** The _data available. */
        private boolean _dataAvailable;

        /** The _read queue. */
        private Vector _readQueue;

        /** The _channel. */
        private int _channel;

        /** The _port. */
        private USBPort _port;

        /* (non-Javadoc)
         * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#init()
         */
        protected void init() throws IOException
        {
          _readQueue = new Vector();

          //register the app for callbacks
          UsbServer.this.addIOPortListener(this);

          /*parent.message("using low level usb interface");*/
          //register the channel
          _channel = USBPort.registerChannel(CHANNEL, 1024, 1024);
          /*parent.message("Registering channel: " + CHANNEL);*/
          synchronized(this)
          {
            try {
              //wait for a channel
              if ( _port == null )
              {
                this.wait();
              }
            } catch (InterruptedException e) {
            }
          }
        }

        /* (non-Javadoc)
         * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#write(java.lang.String)
         */
```

```
protected void write(String s) throws IOException
{
    _port.write(s.getBytes());
}

/* (non-Javadoc)
 * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#read(java.lang.String)
 */
protected String read(String match) throws IOException
{
   try {
      synchronized(this)
      {
         if ( !_dataAvailable )
         {
            this.wait();
         }
         _dataAvailable = false;
      }
   } catch (InterruptedException e) {
      //not thrown by cldc1.0
   }

   //loop through the queue of data and extract each item
   try {
      for(;;)
      {
         byte[] data = null;
         synchronized(this)
         {
            data = (byte[])_readQueue.firstElement();
            _readQueue.removeElement(data); //ensure we always remove the correct object
         }
         String s = new String(data);
         if ( s.equals(match) )
         {
            return s;
         }
      }
   } catch (NoSuchElementException e) {
      //the read queue is empty! we didn't find our match
      /*parent.message("did not recieve match for " + match);*/
   }
   return null;
}

/* (non-Javadoc)
 * @see edu.bristol.crypto.sca.rim.server.UsbServer.UsbThread#close()
 */
public void close()
{
   try {
      /*parent.message("closing connections...");*/
      if ( _port != null) _port.close();
      USBPort.deregisterChannel(_channel); //deregister the channel
      /*parent.message("connections closed");*/
   } catch (IOException e) {
      /*parent.message(e.toString());*/
   }
```

```
}

//usbportlistener methods

/* (non-Javadoc)
 * @see net.rim.device.api.system.USBPortListener#getChannel()
 */
public int getChannel() {
   return _channel;
}

/* (non-Javadoc)
 * @see net.rim.device.api.system.USBPortListener#connectionRequested()
 */
public void connectionRequested() {
   /*message( "lowlevelusb: Connection requested!" );*/
   try {
      _port = new USBPort(_channel);
      synchronized(this) {
         this.notify();
      }
   } catch (IOException e) {
     // message(e.toString());
   }
}

/* (non-Javadoc)
 * @see net.rim.device.api.system.USBPortListener#dataNotSent()
 */
public void dataNotSent() {
   //message( "lowlevelusb: Data not sent" );
}

/* (non-Javadoc)
 * @see net.rim.device.api.system.IOPortListener#connected()
 */
public void connected()
{
  // message( "lowlevelusb: connected" );
}

/* (non-Javadoc)
 * @see net.rim.device.api.system.IOPortListener#disconnected()
 */
public void disconnected()
{
  // message( "lowlevelusb: disconnected" );
}


/* (non-Javadoc)
 * @see net.rim.device.api.system.IOPortListener#receiveError(int)
 */
public void receiveError(int error)
{
  // message( "lowlevelusb: Got rxError: " + error );
}
```

```
    /* (non-Javadoc)
     * @see net.rim.device.api.system.IOPortListener#dataReceived(int)
     */
    public void dataReceived( int length )
    {
      int len;
      DataBuffer db = new DataBuffer();
      try {
        byte[] receiveBuffer = new byte[256];
        if (0 != (len = _port.read( receiveBuffer, 0, length == -1
? receiveBuffer.length : length )) )
        {
          db.write(receiveBuffer, 0, len);
          String data = new String( receiveBuffer, 0, len );
      //      message("lowlevelusb: " + data);
        }
        synchronized(this)
        {
          _readQueue.addElement(db.toArray());
          _dataAvailable = true;
          this.notify();
        }
      } catch( IOException ex ) {
        throw new RuntimeException( ex.getMessage() );
      }
    }

    /* (non-Javadoc)
     * @see net.rim.device.api.system.IOPortListener#dataSent()
     */
    public void dataSent()
    {
    }

    /* (non-Javadoc)
     * @see net.rim.device.api.system.IOPortListener#patternReceived(byte[])
     */
    public void patternReceived( byte[] pattern )
    {
      // message( "lowlevelusb: Got pattern " + new String(pattern));
//+ pattern[0] + " " + pattern[1] + " " + pattern[2] + " " +
//pattern[3] );
    }
  }


  //constructors

  /**
   * Instantiates a new usb server.
   */
  public UsbServer() {

    MainScreen screen = new MainScreen();
    //screen.setTitle(_resources.getFamily(), "USB SERVER");

    _output = new RichTextField("Server Stopped.. Please Run from Menu..");

    screen.add(_output);
```

```
        screen.addKeyListener(this); //implemented by super
        screen.addTrackwheelListener(this); //implemented by super

        pushScreen(screen);
    }

    /**
     * Make menu.
     *
     * @param menu the menu
     * @param instance the instance
     */
    protected void makeMenu(Menu menu, int instance)
    {
            try{
        menu.add(_connectGCF);
//      menu.add(_connectLowLevel);

        menu.addSeparator();
/*      Field focus = UiApplication.getUiApplication().getActiveScreen().getLeafFieldWithFocus();
        if(focus != null) {
          ContextMenu contextMenu = focus.getContextMenu();
          if( !contextMenu.isEmpty()) {
            menu.add(contextMenu);
            menu.addSeparator();
          }
        }
*/      menu.add(_closeItem);

//      makeMenu(menu, instance);
/*      onExit(); //cleanup the old thread if present
        _usbThread = new GCFUsbThread();
        connect();

 */

            }catch (Exception e){

            }
    }

    /**
     * Connect.
     */
    private void connect()
    {
        _usbThread.start();
    }

    /**
     * Message.
     *
     * @param msg the msg
     */
    private void message(final String msg)
    {
        invokeLater(new Runnable() {
          public void run() {
```

```
            _output.setText(_output.getText() + "\n" + msg);
            _output.setCursorPosition(_output.getText().length());
        }
    });
}

/**
 * On exit.
 */
public void onExit()
{
    if ( _usbThread != null ) _usbThread.close();
}


/**
 * To hex.
 *
 * @param b the b
 *
 * @return the string
 */
static public String toHex(byte[] b)  {return toHex(b, b.length);}

/**
 * To hex.
 *
 * @param b the b
 * @param len the len
 *
 * @return the string
 */
static public String toHex(byte[] b, int len)
 {
    if (b==null) return "";
    StringBuffer s = new StringBuffer("");
    int i;
    for (i=0; i<len; i++)
      s.append(toHex(b[i]));
    return s.toString();
}

/**
 * To hex.
 *
 * @param b the b
 *
 * @return the string
 */
static public String toHex(byte b)
{
    Integer I = new Integer((((int)b)<< 24) >>> 24);
    int i = I.intValue();

    if ( i < (byte)16 )
        return "0"+Integer.toString(i, 16);
    else
        return     Integer.toString(i, 16);
}
```

```
/**
 * Invoked when the trackwheel is clicked.
 *
 * @param status the status
 * @param time the time
 *
 * @return true, if trackwheel click
 */
public boolean trackwheelClick( int status, int time )
{
   Menu menu = new Menu();

   makeMenu(menu, 0);

   menu.show();

   return true;
}

/**
 * Invoked when the trackwheel is released.
 *
 * @param status the status
 * @param time the time
 *
 * @return true, if trackwheel unclick
 */
public boolean trackwheelUnclick( int status, int time )
{
   return false;
}

/**
 * Invoked when the trackwheel is rolled.
 *
 * @param amount the amount
 * @param status the status
 * @param time the time
 *
 * @return true, if trackwheel roll
 */
public boolean trackwheelRoll(int amount, int status, int time) {
   return false;
}

/* (non-Javadoc)
 * @see net.rim.device.api.system.KeyListener#keyChar(char, int, int)
 */
public boolean keyChar(char key, int status, int time) {
   //intercept the ESC key - exit the app on its receipt
   boolean retval = false;

   switch (key) {
   case Characters.ESCAPE:
      onExit();
      System.exit(0);
      retval = true;
      break;
```

```
        }

        return retval;
    }

    /**
     * Implementation of KeyListener.keyDown
     *
     * @param keycode the keycode
     * @param time the time
     *
     * @return true, if key down
     */
    public boolean keyDown(int keycode, int time) {
        return false;
    }

    /**
     * Implementation of KeyListener.keyRepeat
     *
     * @param keycode the keycode
     * @param time the time
     *
     * @return true, if key repeat
     */
    public boolean keyRepeat(int keycode, int time) {
        return false;
    }

    /**
     * Implementation of KeyListener.keyStatus
     *
     * @param keycode the keycode
     * @param time the time
     *
     * @return true, if key status
     */
    public boolean keyStatus(int keycode, int time) {
        return false;
    }

    /**
     * Implementation of KeyListener.keyUp
     *
     * @param keycode the keycode
     * @param time the time
     *
     * @return true, if key up
     */
    public boolean keyUp(int keycode, int time) {
        return false;
    }
}
```

USBclient.cpp: modified from original provided by RIM to support sending commands and to integrate with Matlab

```
#include "mex.h"
#import "BBDevMgr.exe" no_namespace named_guids

enum{ READ_EVENT, CLOSE_EVENT, NUM_EVENTS};

typedef struct {
   unsigned short length;
   unsigned char data[256];
} DEVICE_DATA;

static const int HEADERSIZE = sizeof(unsigned short);

//static char const * const PC_HELLO = "HHHH;";
static char const * const PC_GOODBYE = "Goodbye from P;";
static char const * const DEVICE_HELLO = "Hello from Device";
static char const * const DEVICE_GOODBYE = "Goodbye from Device";
int status=-1;
#define CHANNEL L"R"
class USBClient:
   public IChannelEvents //implements the ICHannelEvents interface in order to receive notifications on the connection to the device
{
public:
   USBClient()
   {
   }

   ~USBClient()
   {}

public:

   char * run(char *PlainText)
   {

   //create some events for listening for notifications from the device
   _events[READ_EVENT] = CreateEvent(NULL, FALSE, FALSE, NULL); //unnamed
   _events[CLOSE_EVENT] = CreateEvent(NULL, TRUE, FALSE, NULL); //unnamed
   if( _events[READ_EVENT] == INVALID_HANDLE_VALUE || _events[CLOSE_EVENT] == INVALID_HANDLE_VALUE ) {
      return NULL;
   }

   try {
      CoInitializeEx(NULL, COINIT_MULTITHREADED);

      //grab an instance of the main proxy object: IDeviceManager
      IDeviceManagerPtr deviceManager; //(__uuidof(IDeviceManager));
      deviceManager.CreateInstance(CLSID_DeviceManager);
      //get the list of attached devices
      IDevicesPtr devices = deviceManager->Devices;

      //query the count of attached devices
      if ( devices->Count > 0 )
      {
      IDevicePtr device = devices->Item(0); //get the first
```

```
   //get the properties for the attached device
   IDevicePropertiesPtr properties = device->Properties;
   long count = properties->Count;
   for (long i = 0; i < count; ++i)
   {
      IDevicePropertyPtr property = properties->Item[i]; //(_variant_t(i));

      char name[256];
      memset( name, 0, sizeof( name ) );
      WideCharToMultiByte( CP_ACP, 0, property->Name, SysStringLen( property->Name ),
            name, sizeof( name ), NULL, NULL );

      //printf("Name: %s\n",
      //name);

      _variant_t var(property->Value);
      if ( strcmp("BBPIN", name) == 0 )
      {
      long pin = (long)(var);
      //printf("Value: 0x%x\n", pin);
      }
      else
      {
      //printf("Value: %s\n",
      //(char*)((_bstr_t)var));
      }

   }

   //open a channel and exchange some data
   //printf("");
   //printf("Opening Channel: %s\n", CHANNEL);
   _channel = device->OpenChannel(CHANNEL, this);

   char buffer[258];
   char * pString = NULL;
   char *s;

   long len = 0;

   /*Start Sending & Receiving*/


   strcpy(buffer, PlainText);


   Send(buffer);
   return s;
   }
} catch ( _com_error e) {
   //printf("Connection Error");
   status=-1;
   //ComError(e);
}

return NULL;
}
```

```
private:

   char * ReadVariableProtocol(char const * match, char * buffer, int bufferLen, long & outputLen)
   {
   char * p = NULL;
   if ( Receive((unsigned char*)buffer, bufferLen, outputLen) )
   {
      //figure out if the length bytes are leading
      if ( strncmp(buffer, match, HEADERSIZE) != 0)
      {
      //we're using the GCF, and we have length bytes prepended to the start
      p = (char *)((DEVICE_DATA*)buffer)->data;
      outputLen = outputLen - HEADERSIZE;
      }
      else
      {
      p = buffer;
      }
      //printf("Received: %s\n", p);
   }
   return p;
   }


   bool Send(char * buf)
   {
      //printf("Sending: %s\n", buf);
      HRESULT result = _channel->WritePacket((unsigned char*)buf, strlen(buf));
      return ( !FAILED(result));
   }

   bool Receive(unsigned char *buf,char terminator,long &lengthReceived)
   {
      char *temp;
      bool flag = false;
      buf='\0';
      int counter=0;
      while(!flag){
         switch( WaitForMultipleObjects( /*NUM_EVENTS*/0, _events, FALSE, 1000 ) ) { //30 sec timeout
         case WAIT_OBJECT_0:
         try {
            //printf("\nlength of buf = %d",counter);
            HRESULT result = _channel->ReadPacket( buf+(counter++), 1, &lengthReceived );
            //printf("\nreceived %s",buf);
            if(buf[counter-1]==terminator){
               flag=true;
            }
            return( !FAILED( result ) );
         } catch( _com_error e ) {
            //ComError( e );
            status=-1;
            return false;
         }
         break;
         case WAIT_OBJECT_0 + 1:
         // Closed
         //printf("Port closed\n" );
         return false;
         case WAIT_TIMEOUT:
```

```
        //printf("Timeout waiting for inbound data\n");

        return false;
        default:
        // Shouldn't happen
        return false;
    }
  }
  }


  bool Receive(unsigned char * buf, int len, long & actual)
  {
  switch( WaitForMultipleObjects( NUM_EVENTS, _events, FALSE, 1000 ) ) { //30 sec timeout
    case WAIT_OBJECT_0:
    try {
      HRESULT result = _channel->ReadPacket( buf, len, &actual );

      return( !FAILED( result ) );
    } catch( _com_error e ) {
      //ComError( e );
      status=-1;

      return false;
    }
    break;
    case WAIT_OBJECT_0 + 1:
    // Closed
    //printf("Port closed\n" );
    return false;
    case WAIT_TIMEOUT:
    //printf("Timeout waiting for inbound data\n");
    status=0;
    return false;
    default:
    // Shouldn't happen
    return false;
  }
  }

  void Close()
  {
  //close the channel and wait for BBDevMgr to shut it down
  _channel = NULL;

  // Wait for BbDevMgr to close the channel, otherwise it ends up crashing.
  WaitForSingleObject( _events[CLOSE_EVENT], 5000 );

  CloseHandle( _events[READ_EVENT] );
  CloseHandle( _events[CLOSE_EVENT] );
  }

  void ComError(_com_error e)
  {
  ::MessageBox(NULL, e.ErrorMessage(), "USBClient ERROR", MB_ICONERROR | MB_OK );
  }


public:
```

```
  //implementation of the IChannelEvents interface
  virtual HRESULT STDMETHODCALLTYPE QueryInterface( REFIID riid, void **ppvObject )
  {
  if ( riid == IID_IChannelEvents || riid == IID_IUnknown) {
    *ppvObject = (IChannelEvents*)this;
    AddRef();
    return 0;
  }
  return E_NOINTERFACE;
  }

  virtual ULONG   STDMETHODCALLTYPE AddRef()
  {
  return ++_refcount;
  }

  virtual ULONG   STDMETHODCALLTYPE Release()
  {
  ULONG count = --_refcount;
  if ( count == 0 )
  {
    delete this;
  }
  return count;
  }

  virtual HRESULT STDMETHODCALLTYPE raw_CheckClientStatus( void )
  {
  return 0; //not implemented
  }

  virtual HRESULT STDMETHODCALLTYPE raw_OnChallenge(
    /* [in] */ long attemptsRemaining,
    /* [out] */ unsigned char passwordHash[20] )
  {
  attemptsRemaining;
  passwordHash;



#if defined(_DEBUG)
  _asm {
    int 3; //insert a breakpoint
  }
#endif

  return E_FAIL;

  /*
  const char * password = "password";
  sha_hash(password, strlen(password), passwordHash);
  return S_OK;
  //*/
  }

  virtual HRESULT STDMETHODCALLTYPE raw_OnNewData( void )
  {
  SetEvent(_events[READ_EVENT]);
  return 0;
```

```
    }

    virtual HRESULT STDMETHODCALLTYPE raw_OnClose( void )
    {
    SetEvent(_events[CLOSE_EVENT]);
    return 0;
    }

private:
    //data members
    IChannelPtr _channel;
    ULONG _refcount;
    HANDLE _events[NUM_EVENTS];
};

int main(int argc, char* argv[])
{
    argc;
    argv;
    USBClient usb;

}

/* The gateway function */
void mexFunction( int nlhs /* # of outputs*/, mxArray *plhs[],
                  int nrhs /* # of inputs*/, const mxArray *prhs[])
{
/* variable declarations here */
    const mxArray *yData;
    int yLength;
    char *TheString;


    //Copy input pointer y
    yData = prhs[0];

    //Make "TheString" point to the string
    yLength = mxGetN(yData)+1;
    TheString = (char *)mxCalloc(yLength, sizeof(char)); //mxCalloc is similar to malloc in C
    mxGetString(yData,TheString,yLength);
    //printf("Plain Text: [%s]\n",TheString);

    //strcpy(PC_HELLO,TheString);
    //PC_HELLO=TheString;

    USBClient usb;
    usb.run(TheString);

//    printf("status: %d\n",status);
//    nlhs=1;
//    plhs[0] = (mxArray *)status; //mxReal is our data-type
}
```

Matlab script used to acquire AES recordings: This is a modified version of an OpenSCA acquire script originally used to trigger encryptions on ARM7 CPU.

```
clear all;

if (exist('CONTAINER_AES_PW_Traces', 'dir'))
  rmdir('CONTAINER_AES_PW_Traces','s');
end

display('Initializing oscilloscope...');
osci = osci_dpo7254(1e7,1e7,1); %OK

% create container for traces and results
tc_pw_name = 'AES_PW_Traces'; %OK
tc_pw_dir = '.'; %OK
tc_pw_desc = 'Power traces of AES encryption operations acquired using the ARM setup and while using EM probe with label B. Configuration file for the scope is stored in the CONFIG directory. 47 Ohm resistance.';
tc_pw_nickname = 'AES ARM Traces';
TraceContainer = container(tc_pw_name, tc_pw_dir, tc_pw_desc, tc_pw_nickname,0,1); % OK

[offset, scale, sample_rate] = get_trace_settings(osci, 'CH3'); % CH3?

i = 1;
num_encryptions = 1000;

InputContainer = container('AES_plaintext','.');
% read in inputs
register_input = get(InputContainer,'ContainerElements',1:num_encryptions);

while(i <= num_encryptions)
   display(sprintf('Executing AES %05d/%d', i, num_encryptions));

   % Arm the scope's trigger
   arm_nb(osci); %OK

   pause(1);
   to_send =
sprintf('ENCRYPT:ALGORITHM,AES:KEY,fedcba98765432100123456789abcdef:PLAINTEXT,%s%s%s%s
:TRIGGER,LED;',
dec2hex(register_input(i,1),8),dec2hex(register_input(i,2),8),dec2hex(register_input(i,3),8),dec2hex(regi
ster_input(i,4),8));
   display(sprintf('BB Command [%s]',to_send));
   testMexFNF(to_send)

   % Block Matlab until the scope returns a ready signal
   % or a timeout. If an timeout occurs the input data
   % should normally be processed again. So it is recommended
   % not to increment the loop counter and to continue from the
   % beginning of the loop.
   pause(1);
   timeout = block(osci);
   if(timeout)
      continue;
   else
%       res = read_small(osci, 'CH1',0,0.55);
%       size(res);
%       add(TraceContainer_pw, res');
```

```
      res = read_small(osci, 'CH3',0.15,0.45);
      size(res);
      add(TraceContainer, res');

     i = i + 1;
   end
end

% fclose(ser);
delete(osci);
% flush(TraceContainer);
flush(TraceContainer);
```

Automatic Alignment Script:

```
clear all;
decimationFactor=80;
flag=0; %0=create, 1=add
if flag==1
PartialPWContainer = container('AES_PARTIAL', '.');

else

   if (exist('CONTAINER_AES_PARTIAL', 'dir'))
     rmdir('CONTAINER_AES_PARTIAL','s');
   end
   tc_pw_name = 'AES_PARTIAL'; %OK
   tc_pw_dir = '.'; %OK
   tc_pw_desc = 'PARTIAL AES Traces TEST';
   tc_pw_nickname = 'PARTIAL AES ARM Traces';
   PartialPWContainer = container(tc_pw_name, tc_pw_dir, tc_pw_desc, tc_pw_nickname,0,1); % OK
end
i=1;
run=10;
num_encryptions = run+i-1;

InputContainer = container('AES_plaintext','.');
PWContainer = container('AES_PW_Traces', '.');

register_input = get(InputContainer,'ContainerElements',1:num_encryptions);
register_PW = get(PWContainer,'ContainerElements',1:num_encryptions);

ref_signal=decimate(register_PW(1,1:750000),decimationFactor);
while(i <= num_encryptions)

   display(sprintf('Aligning AES Trace %05d/%d', i, num_encryptions));

 display(sprintf('BB Command [%s]',to_send));
temp=register_PW(i,1:750000);

   start_index=0;
   end_index=0;
   flag=0;
      for w=100000:1:190000
         for e=1:1:20
            if temp(w)>40 && temp(w+e+10863-10)>40
               for r=10:1:10800
                  if temp(r+w)>40
                     flag=1;
                     display('hi');
                     break;
                  end
               end
               if flag==0
                  start_index=w;
                  display(start_index);
               end
               flag=0;
               break;
            end
            if start_index>0
               break
```

```
            end
        end
        if start_index>0
            break
        end
    end

add(PartialPWContainer,circshift(temp,[1,172911-start_index]));
i=i+1;
end
```

Arff creator: used to create Weka compatible training files.

```
clear all;
i=1;
run=50;
num_encryptions = run+i-1;
training_rounds=1;
InputContainer = container('AES_plaintext','.');
AlignedContainer = container('AES_PARTIAL', '.');
register_input = get(InputContainer,'ContainerElements',1:num_encryptions);
register_PW = get(AlignedContainer,'ContainerElements',1:num_encryptions);

points=[217190,266600,315370,363980,409210,455460,500850,542550,586180];
%points(1)=147140; for first round only
fragmentSize=22996;
%fragmentSize=16000; for first round only

fileId=fopen('AES.arff','w+');
fprintf(fileId,'@RELATION AESFragments\n');
fileId2=fopen('meanSd.txt','w+');

for k=1:1:2
   fprintf(fileId,'@ATTRIBUTE %d numeric\n',k);
end
fprintf(fileId,'@ATTRIBUTE CLASS {firstHalf,secondHalf}\n@DATA\n');

while(i <= num_encryptions)
   temp =  register_PW(i,1:750000);
   for m=1:1:training_rounds
      frag1= temp(points(m)-fragmentSize:points(m));
      sum=0;
      for j=1:1:fragmentSize
         sum=sum+temp(points(m)-fragmentSize+j);
      end
      mean=sum/fragmentSize;
      dif=0;
      for j=1:1:fragmentSize
         dif=dif+(temp(points(m)-fragmentSize+j)-mean)^2;
      end
      stdDiv=(dif/fragmentSize)^1/2;
      fprintf(fileId,'%f,%f,',mean,stdDiv);
      fprintf(fileId,'firstHalf\n');
      fprintf(fileId2,'%f\t%f\n',mean,stdDiv);
      frag2= temp(points(m):points(m)+fragmentSize);
      sum=0;
      for j=1:1:fragmentSize
         sum=sum+temp(points(m)+j);
      end
      mean=sum/fragmentSize;
      dif=0;
      for j=1:1:fragmentSize
         dif=dif+(temp(points(m)+j)-mean)^2;
      end
      stdDiv=(dif/fragmentSize)^1/2;
      fprintf(fileId,'%f,%f,',mean,stdDiv);
      fprintf(fileId,'secondHalf\n');
   end
   i=i+1;
end
```

BuildClassifierEM: Used to train and evaluate classifiers

```
package weka.train;

import java.awt.BorderLayout;
import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.ObjectOutputStream;
import java.util.Random;
import weka.classifiers.Classifier;
import weka.classifiers.Evaluation;
import weka.classifiers.bayes.NaiveBayes;
import weka.classifiers.bayes.NaiveBayesMultinomial;
import weka.classifiers.evaluation.ThresholdCurve;
import weka.classifiers.functions.MultilayerPerceptron;
import weka.classifiers.functions.SVMreg;
import weka.classifiers.lazy.IB1;
import weka.classifiers.meta.FilteredClassifier;
import weka.core.Instances;
import weka.core.Utils;
import weka.gui.visualize.PlotData2D;
import weka.gui.visualize.ThresholdVisualizePanel;
public class BuildClassifierEM {

	/**
	 * @param args
	 */

	public static void main(String[] args) throws Exception {
		    // load data

		Instances data = new Instances(new BufferedReader(new
FileReader("./arff/AES.arff")));
		data.setClassIndex(data.numAttributes() - 1);

		    // train classifier

		Classifier cl;

		cl = new NaiveBayes();
		cl=  new MultilayerPerceptron();
		cl.buildClassifier(data);
		System.out.println(cl.getCapabilities().getOwner());
	   //FileOutputStream fstream = new FileOutputStream("classifier.xml");
		FileOutputStream fos = new FileOutputStream("AES.dat");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(cl);
/*
		cl = new FilteredClassifier();
		cl.buildClassifier(data);
		System.out.println(cl.getCapabilities().getOwner());
	   //FileOutputStream fstream = new FileOutputStream("classifier.xml");
		fos = new FileOutputStream("Filtered.dat");
		oos = new ObjectOutputStream(fos);

	*/
	   // Write object.
```

```
            oos.writeObject(cl);


            //cl=new NaiveBayes();
//cl=new BayesN;

            Evaluation eval = new Evaluation(data);
            //eval.evaluateModel(arg0, arg1)

            eval.crossValidateModel(cl, data, 10, new Random(1));
            int halves=2;
            int score[]=new int[100];
            int count=0;
            int r[]=new int[halves];
            Random random=new Random();
            for(int i=0;i<data.numInstances();i++){
                  double d=cl.classifyInstance(data.instance(i));
                  double []dist=cl.distributionForInstance(data.instance(i));
              r[i%halves]=dist[0]/dist[1]>1?1:-1;
                  if(i%halves==0&&i>0){
                        for(int j=0;j<halves;j++){
                              score[i/halves]+=j%2==0?r[j]:(-r[j]);
                        }
                        count=i/halves;
                  System.out.println(score[i/halves]);
              }


    System.out.println((i+"\t"+dist[0]/dist[1])+"\t"+(dist[0]/dist[1]>1?"1st":"2nd"));

            }
            double sum=0;
            for(int j=0;j<count;j++){
                  sum+=score[j];
            }
            double mean=sum/count;
            sum=0;
            for(int i=0;i<count;i++){
                  sum+=Math.pow(score[i]-mean,2);
            }
            System.out.println("MEAN = "+mean+" sd = "+ Math.pow(sum,0.5)/count);

               // generate curve

               ThresholdCurve tc = new ThresholdCurve();
               int classIndex = 0;
               System.out.println(eval.toSummaryString());

               Instances result = tc.getCurve(eval.predictions(), classIndex);

               // plot curve
               ThresholdVisualizePanel vmc = new ThresholdVisualizePanel();
               vmc.setROCString("(Area under ROC = " +
                 Utils.doubleToString(tc.getROCArea(result), 4) + ")");
               vmc.setName(result.relationName());
               PlotData2D tempd = new PlotData2D(result);
               tempd.setPlotName(result.relationName());
               tempd.addInstanceNumberAttribute();
               vmc.addPlot(tempd);
```

```
                System.out.println(result.toSummaryString());

                // display curve

                String plotName = vmc.getName();
                final javax.swing.JFrame jf =
                  new javax.swing.JFrame("Weka Classifier Visualize: "+plotName);
                jf.setSize(1000,800);
                jf.getContentPane().setLayout(new BorderLayout());
                jf.getContentPane().add(vmc, BorderLayout.CENTER);
                jf.addWindowListener(new java.awt.event.WindowAdapter() {
                  public void windowClosing(java.awt.event.WindowEvent e) {
                  jf.dispose();
                  }
                });
                jf.setVisible(true);

        }
}
```

BB Commands User Documentation:

BB commands sent through USB interface offer access to AES and DES block ciphers, and are easily extensible as it follows a token based format.

General Syntax:

COMMAND_ID:PARAMTER1_NAME,PARAMTER1_VALUE:PARAMTER2_NAME,PARAMTER2_VALUE: … :PARAMTERN_NAME,PARAMETERN_VALUE;

Here special characters which should be avoided other than to provide the functionality below are:

Special character	Description
:	Used to separate parameters
,	separates a parameter name from its value
;	indicates a command has terminated

Encrypt Command:

Syntax:

ENCRYPT:ALGORITHM,algorith:PLAINTEXT,plaintext:KEY,key:TRIGGER,trigger:PREWAIT,prewait:POSTWAIT,postwait:RESPOND,responseFlag;

Parameters:

Name	**Description**	**Values**	**Optional**
Algorithm	Encryption Algorithm to be used	1) AES 2) DES	NO
Key	Key to be used by algorithm in Hex	1) 128-bit AES 2) 192-bit AES 3) 256-bit AES 4) 56-bit DES In DES pass 64 bits, the lsb of every byte is discarded	YES If not passed: 1) First invocation, random key generated per algorithm 2) Last valid key used otherwise
Plaintext	Plaintext in a chosen plaintext attack in Hex format	Any Hex Value. Multiple encryptions in ECB mode will be triggered if plaintext is longer than block	NO

		size	
Trigger	Trigger is switched on before encryption and off after	LED, BACKLIGHT	NO
Prewait	Wait period after trigger and before encryption takes place	Integer value in milliseconds	YES
Postwait	Wait period after encryption and before trigger is turned off	Integer value in milliseconds	YES
Respond	For either synchronous or asynchronous execution	YES or NO	YES Default value is NO (asynchronous)

NB: No branching is performed after trigger. Separate functions written for every scenario. In Matlab all commands are fire and forget. Separate client has also been modified to display response. Response is a simple HEX string containing result from the operation. In this instance response will be the cipher text.

Examples:

DES Command (illustrating ECB mode):
ENCRYPT:ALGORITHM,DES:PLAINTEXT,0123456789abcdef0123456789abcdef:KEY,0123456789abcdef:TRIGGER,LED:RESPOND,YES;

Response:
56cc09e7cfdc4cef56cc09e7cfdc4cef

DES Command (illustrating ignored bits):
ENCRYPT:ALGORITHM,DES:PLAINTEXT,0123456789abcdef:KEY,0022446688aaccee:TRIGGER,LED:POSTWAIT,30:RESPOND,YES;

Response:
56cc09e7cfdc4cef

AES Command (128-Bit):
ENCRYPT:ALGORITHM,AES:PLAINTEXT,0123456789abcdeffedcba9876543210:KEY,0123456789abcdeffedcba9876543210:TRIGGER,LED:PREWAIT,30:RESPOND,YES;

Response:
a674f5a389253565260d08dcbed5c971

AES Command (192-Bit):
ENCRYPT:ALGORITHM,AES:PLAINTEXT,0123456789abcdeffedcba9876543210:KEY,0123456789abcdeffedcba98765432100123456789abcdef:TRIGGER,LED:PREWAIT,30:POSTWAIT,30:RESPOND,YES;

Response:
4b40b6f939cf3cc95487797ff3169f78

AES Command (256-Bit):
ENCRYPT:ALGORITHM,AES:PLAINTEXT,0123456789abcdeffedcba9876543210:KEY,0123456789abcdeffedcba98765432100123456789abcdeffedcba9876543210:TRIGGER,LED:PREWAIT,30:POSTWAIT,30:RES

POND,YES;

Response:
65561e88a83c44dbb99c18d9a63e5620

Appendix B: Poster