# A Novel Counter System against Power Analysis Attacks

Mr. Harshal B Torvi
Computer Science and Engineering Dept
VVPIET, Solapur, India.
harshal.torvi@gmail.com

Mr. Shambhuraj Deshmukh
Computer Science and Engineering Dept
VVPIET, Solapur, India.
deshmukhsp27@gmail.com

Mr. Vishwakarma Vijaykumar Panchal
Computer Science and Engineering Dept
KECSP, Solapur, India
pvishwa301@gmail.com
9404522405

**Abstract**— Many cryptographic algorithms are vulnerable against various attacks including side channel attacks. A side channel attack causes a continues leakage of information through the physical structure of a cryptosystem. For example, a large delay in information retrieval, heavy power consumption, which can generate damage to the system. This paper introduces a compiler to protect cryptographic algorithms against various attacks by executing certain suits of tasks automatically. Firstly the compiler detects the most sensitive instructions which leak the most information during side channels attack. This process is accomplished either by static or dynamic analysis. In dynamic analysis, metric in terms of information theory is acquired over the power traces during the completion of the input program, while in static analysis, as information leakage starts implying in a loss of security, the compiler produces a batch of instructions with a software countermeasure such as random precharging or Boolean masking. This software protection will results in significant overhead in terms of time and space complexity of cryptographic algorithm. The proposed compiler is crosschecked against two block ciphers algorithms, AES and Clefia; and experimentally it is clear that the compiler offers best productivity for cryptosystem developers to protect their implementations from side channel attacks.

**Keywords-** SIDE-CHANNEL ATTACKS, POWER ANALYSIS ATTACKS, SOFTWARE COUNTERMEASURES, COMPILER

## 1 INTRODUCTION

SECURITY is a fundamental criteria in most of today's computing domains That means hardware and software systems must be designed with extreme security to restrict attackers to access the confidential data. Presently, hardware and software tools do not emphasis on security, despite of its major importance. For instance, side channel attacks are a major area of consideration. As enumerated in abstract, side channel attacks harm the physical layer of a cryptographic algorithm, keeping the internal mathematical structure of the system intact. Some of publicly known side channels attacks are a large delay in information retrieval, heavy power consumption [5] etc. these attacks are performed by an adverse accessing the device, and encrypting finite sample of plaintexts, without the exact secret key used in the device. Lots of suits of tasks (we call it as countermeasures) are suggested against these side channel attacks; typically these countermeasures are manually designed by experts to demolish the effect of side channel attack. So to free from the dependency of the expert, we propose a compiler which automatically produces some software countermeasures for cryptographic system to prevent it from side channel attacks.

These countermeasures may produce effect on the performance and code size of input program; also it is not practical to protect every instruction in a cryptographic algorithm. Therefore, to address the above problems, proposed compiler recognizes a subset of the instructions for protection, subject to meeting a desired level of security.

The flow of compilation process to protect against side channel attack constitutes following three steps:

- Information Leakage Analysis predicts the instructions which are vulnerable to side channel attacks. The compiler can find out measurements regarding side channel attack via power traces during static analysis.
- Transformation Target Identification finds data dependencies within the sensitive instructions and thereby larger groups of instructions that deserves the security.
- Code Transformation executes mechanism on sensitive instruction. (See Fig 1)

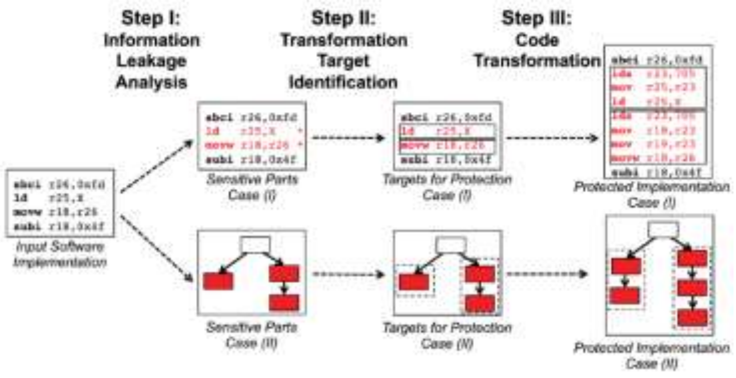

Step I: Information Leakage Analysis
Step II: Transformation Target Identification
Step III: Code Transformation
Input Software Implementation
Sensitive Parts Case (I)
Targets for Protection Case (I)
Protected Implementation Case (I)
Sensitive Parts Case (II)
Targets for Protection Case (II)
Protected Implementation Case (II)

**Fig:1 Three Stages of Compiler[1]**

That means priory, compiler predicts the sensitive instructions that looses the most information.

In next step, the compiler groups the deserved instructions to stop information leakage. Finally, some countermeasure are executed on instructions to generate the transformed program which is the secure enough against any kind of attack.The proposed compiler can be evaluated on block cipher algorithms like, AES and Clefia, using software countermeasures like random precharging and Boolean masking. Random precharging requires local analysis and code transformations; while Boolean masking requires global form of treatment. The remaining paper is structured as follows. Section 2 discusses the related work. Sections 3 to5 respectively describe the three key steps of proposed compiler. Section 6 outlines experimental setup regarding this project . Finally section 7 concludes the paper.

## 2 RELATED WORK

Traditionally, cryptanalytic attacks were damaging mathematical structure of cryptographic algorithms keeping physical level of implementation intact. Contrast to it, side channel attacks work with different way: instead of mapping binary inputs with binary outputs, side channel attackers destroys the relations between this information with consuming excess power to uncover the secret key. These attackers need not require specific knowledge regarding the internal details of the device. The attacker must aware of only the algorithm being executed during the attack. The proposed compiler enhances the functioning of algorithm against power based side-channel attacks.

In network security, power analysis attack is a form of side channel attack in which the attacker records the power consumption of a cryptographic hardware devices like smart card, integrated circuit etc. and thereby retrieve secret keys and other confidential information from the device. Simple power analysis (SPA) is a side channel attack which visually examines power graph of a device over time. And records variations in power consumption as device performs various operations.

Most of the mechanization existing for side channel protection concentrate on hardware countermeasures. For example, Tiwari et al. [9] introduced secure power-analysis-attack-resistant ICs which adopt a digital very large scale integrated (VLSI) design flow.

While performing the review on this work, Moss et al. [7] proposed Boolean masking technique. But this method didn’t take care about the sequence of instruction execution in the output that’s why it leads to susceptible codes in most of today’s devices. So ultimately, this technique fails if unique mask is used in two consecutive instructions. On the other hand, our compiler transforms assembly language instructions and avoids susceptible orderings. Secondly, there are lots of formatting concerns on high-level source code in the work proposed by Moss et al. This is also not the issue in working of our compiler.

## 3 PROPOSED WORK

The proposed compiler framework totally based on three important steps. In this section, the first step of compilation process is clearly examined

## 3.1 ANALYSIS OF INFORMATION LEAKAGE

This is the first step which determines instructions leaking the most sensible information causing side-channel attack. The cryptographic algorithm in terms of assembly language instructions is fed as an input to this step for the protection, and the output is produced in terms of comments describing the sensitivity of each instruction.

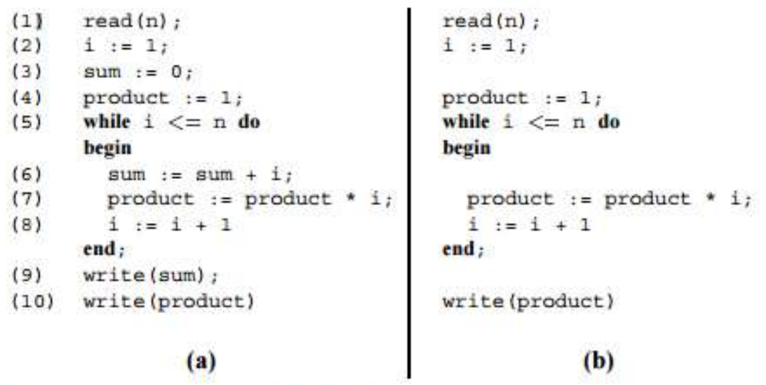
The reason that we choose to operate on assembly The reason behind feeding assembly instructions as an input is to preserve the output behavior of the program by applying the countermeasures including detection of redundant code, and eliminate them so as to improve performance and reduce code size. That means if higher level, instructions are added in the form of countermeasure, it would not harm the output of the program. Random precharging is the good example of this process.

There are two ways to perform information leakage including static analysis , dynamic analysis. In static analysis, assembly code is decompiled into a traditional intermediate representation by using program slicing techniques[9] and thereby sensitive instructions along with their dependencies, are also identified.

While in Dynamic Analysis, power traces are provided by the user and dynamically sensitive instructions are identified with it.

### 3.1.1 Program Slicing techniques

A Program Slicing techniques emphasis on obtaining parts of a program (i.e program slice) from the given program based on slicing criterion. It is actually an algorithm by which a Program P is reduced to slice S by applying certain criteria C. In other words it can be termed in terms of function S=f(P,C) where f is algorithm, that uses P as the input program and obtains slice S from it by applying slicing criteria C.



```
(1)    read(n);
(2)    i := 1;
(3)    sum := 0;
(4)    product := 1;
(5)    while i <= n do
       begin
(6)      sum := sum + i;
(7)      product := product * i;
(8)      i := i + 1
       end;
(9)    write(sum);
(10)   write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

Figure 1: **(a)** An example program. **(b)** A slice of the program w.r.t. criterion (10, product).

Figure 1 (a) shows a sample program that takes a number n, and calculates the sum and the product of first n positive numbers. While figure 1 (b) is the reducible slice of this program based on criterion (10, product) which includes the code regarding the product of first ten positive integer only. Returning to the discussion, details regarding Static and Dynamic Analysis are as follows.

### 3.1.2 Static Analysis

In this type of analysis, given cryptographic software is automatically analyzed and sensitive instructions are correctly highlighted. At very first, the source program is decompiled into a Control Flow Graph (CFG), along with a Data Flow Graph (DFG) to symbolize dependencies. Various decompilation methods are summarized in the Ph.D. thesis of Cifuentes [9], which is shown in Fig. 2.

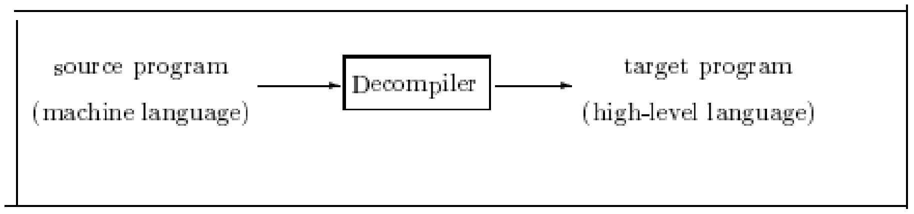


Fig 2: Decompiler[9]

A Decompiler is a program that reads the source code in machine languages and translate it into high level language i.e. it works exactly in reverse manner to that of compiler which translates high level program into object files . The process of decompiling is also having some finite number of steps which are represented in diagram next(see fig 3).

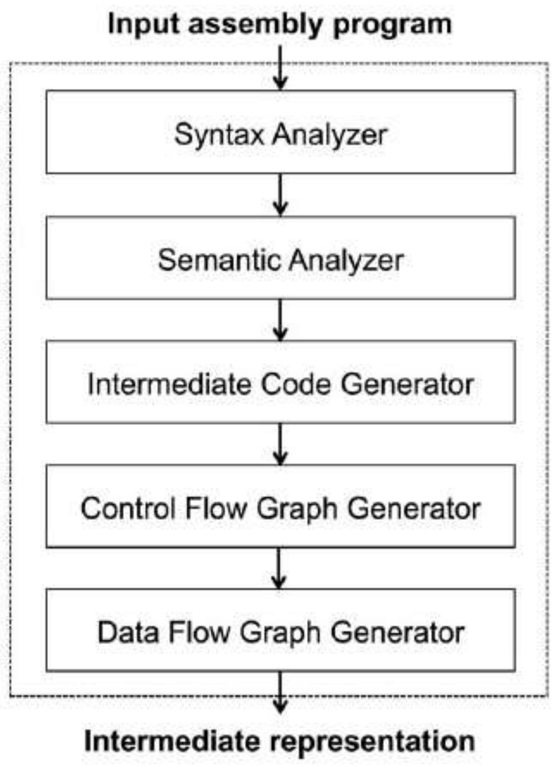

Input assembly program
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Control Flow Graph Generator
Data Flow Graph Generator
Intermediate representation

Fig 3: Steps of Decompiler [9]

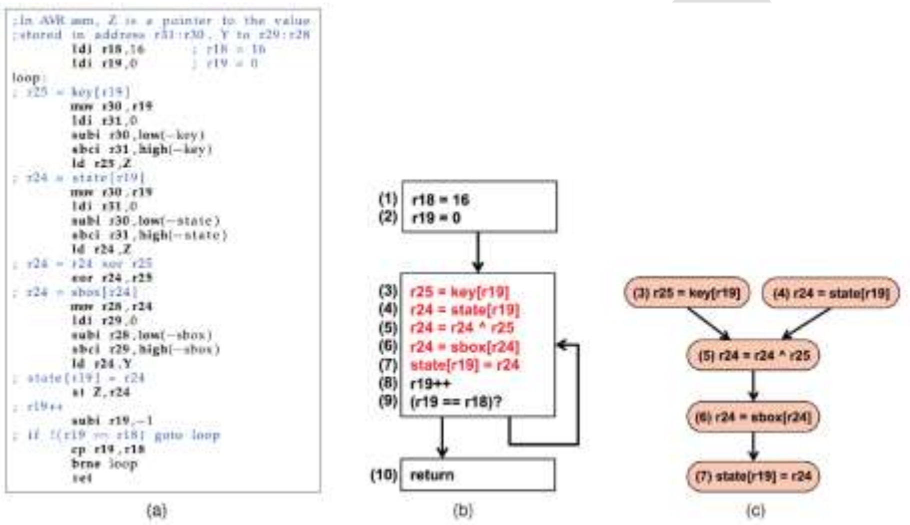
Returning to the discussion, during static analysis, generated intermediate code composed of CFG and DFG instead high-level language representation of the application there by avoids high level code optimizations and code generation phases. The decompiler can be implemented in C++ using Lex and Yacc [2].

An idiom is a set of instructions executing a single operation. For example, a set of instructions used by 8-bit processor to complete a 16-bit operation, which can be termed [9]. The compiler builds a basic blocks containing a single idiom. Thus maintains the control structure of the application satisfying control flow paradigm, and the DFG propagates the relevant information regarding the sensitivity level and protection requirements etc. through an idioms. So next, the compiler starts the construction of the DFG as a graph G=(V,E) where V is the node of G and E represents the flow of information through these nodes. If V uses the undesired data, then V is distinguished as sensitive idioms. Moreover its descendents ( if any) are also fixed as sensitive.

The user passes critical variables through command-line to the compiler. However, if user hasn't provided any specific critical variables then compiler by default assumes that all variables from the program are critical.

To motivate above concept, if load operation is taken into the consideration, it copies a byte of the secret key from memory to a register and thereby accesses critical data and is termed as sensitive.

Fig. 4 provides a complete process of static analysis. At first, the compiler examines the idioms and builds the CFG and DFG; afterwards, the DFG is examined to determine the sensitive instructions.



```
;In AVR asm, Z is a pointer to the value
;stored in address r31:r30, Y to r29:r28
        ldi r18,16          ; r18 = 16
        ldi r19,0           ; r19 = 0
loop:
; r25 = key[r19]
        mov r30,r19
        ldi r31,0
        subi r30,low(-key)
        sbci r31,high(-key)
        ld r25,Z
; r24 = state[r19]
        mov r30,r19
        ldi r31,0
        subi r30,low(-state)
        sbci r31,high(-state)
        ld r24,Z
; r24 = r24 xor r25
        eor r24,r25
; r24 = sbox[r24]
        mov r28,r24
        ldi r29,0
        subi r28,low(-sbox)
        sbci r29,high(-sbox)
        ld r24,Y
; state[r19] = r24
        st Z,r24
; r19++
        subi r19,-1
; if !(r19 == r18) goto loop
        cp r19,r18
        brne loop
        ret
```

Fig 4: Static Analysis applied to first two operations of AES[1].

Figure 4 (a) is the source code of first two rounds of AES algorithms. This involves Ex-Or operations of key and state. While figure 4 (b) is the control flow representation of previous source code. The instructions highlighted in red fonts are sensitive instructions. For such sensitive instructions Figure 4 (c) will plot Data Flow graph DFG by interleaving their dependencies.

## 3.1.3 Dynamic Analysis

Dynamic analysis decides the sensitivity of every instruction based on empirical measurements. The users are expected to provide an assembly language implementation of the cryptographic algorithm along with power traces requirement to execute different (plaintext, key) pairs. The user specifies a filename as command line parameter to perform dynamic analysis on it. Based on power requirements, sensitive instructions are found out.

Prior to compiler, power requirement of each instruction is obtained via an oscilloscope; more explanation regarding it is summarized in section 6.1. All these measurements are taken at a high frequency (e.g., 4 GHz in the setup), and then compressed to single power trace for each clock cycle.

During dynamic analysis, compiler performs the following steps automatically:

- The power trace of each instruction is examined, and based on the information theoretic metric, sensitivity of each clock cycle is determined. Information Theory also estimates the amount of information leakage through the system. Clock cycles with power trace exceeding certain threshold set by user, are diagnosed as sensitive instruction. The user can pass the threshold through the command line to the compiler (Ex threshold = val), where val is in the range [0,1] where 0 implies full protection and 1 argues for no protection

- Next the compiler maps each clock cycle from the traces with an assembly instruction. Therefore most sensitive clock cycle are automatically mapped with sensitive instruction.

However, it should be noted that, the sensitivity of the entire system is dictated by its most sensitive clock cycle. Fig. 5 denotes some sample power traces.

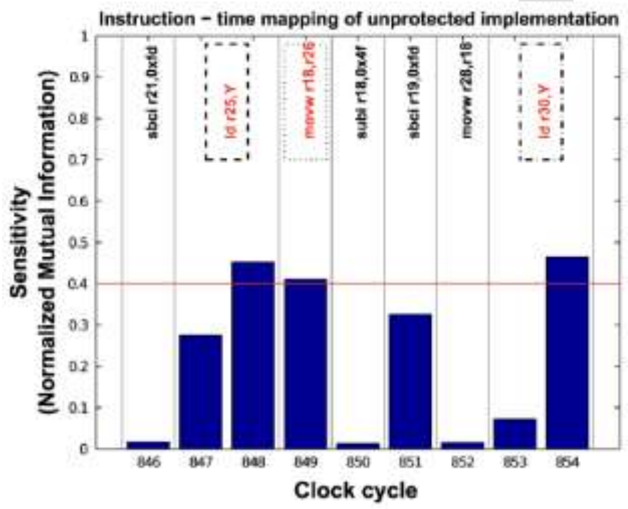


Fig 5: Histogram to devise the sensitive clock cycles [1]

As shown in figure, a sensitivity value is recorded for each cycle, and instructions whose sensitivity exceeds the threshold (0.4 in this example) are diagnosed as sensitive.

Sensitivity Evaluation Estimates:

A metric to evaluate sensitivity is totally based on an information theory expressed by Standaert et al. [3**]**, which measures the resistance of a crypto instructions against the strongest possible power analysis attack. This metric forms a relationship between the secret key by which encryption is done and the power traces per clock cycle.

Let K , X, and L are random variables denoting the secret key, plaintext, and information leakage respectively, and *k*, *x* and *l* be their absolute realizations in the execution of the algorithm. Let μ and σ be the mean and standard deviation of information leakage L when it is uniformly distributed. From above errata, we can state the probability density function for L.

$$N_l(\mu_{k,x}, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(l-\mu_{k,x})^2}{2\sigma^2}},$$

Fig 6: Probability Density Function for Information Leakage [1].

Where μ is average noiseless mean when (*k*, *x*) pair is executed and σ represents standard deviation. The sensitivity of instruction could also be represented in terms of entropy as

$$H[K|L] = -\sum_k p(k) \cdot \sum_x p(x) \cdot \int p(l|k,x) \cdot log_2 p(k|l,x) dl,$$

Fig 7: Entropy of K given L [1]

As shown in figure 7, Entropy to measure the impurity in Secret Key K due to leakage L is calculated. It first considers the probability of proper functioning of key as well as plaintext. Then it considers the probability of leakage l occurs in encryption of plaintext with secret key k.

Which can be rewritten as

$$H[K|L] = -\sum_k \left\{ p(k) \cdot \sum_x \{p(x) \cdot \int_{-\infty}^{\infty} \left\{ N_l(\mu_{k,x}, \sigma^2) \cdot log_2 \frac{N_l(\mu_{k,x}, \sigma^2)}{\sum_{k^*} N_l(\mu_{k^*,x}, \sigma^2)} \right\} dl \right\} \right\}.$$

Fig 8: Entropy of K given L in terms of probability density function [1]

The actual sensitivity of instruction with respect to the Leakage in the key is determined as :

*I [K; L] =H [K]-H [K/L]* *(1)*

In above equation, *I [K; L] denotes* the sensitivity of instruction I with respect to the leakage in secret key K where as *H [K] denotes* entropy to calculate the purity of secret key K, and

*H [K/L] defines* the impurity in secret key K due to leakage L.

## 4 TRANSFORMATION TARGET IDENTIFICATION

Once sensitive instructions are dug out, the compiler automatically chooses instructions for protection. To do this work, some fairly simple countermeasures, including random precharging and random delay insertion, a peephole optimization suffices are available.

With the help of any of above countermeasure, it is possible to protect each sensitive instruction easily, without effecting other instructions in the program. This phase totally emphasis on finding out the sensitive instructions in the cryptographic algorithm. Other types of countermeasure like masking and instruction shuffling protect idioms relying on critical data.

The transformations which we apply to an every idiom heavily rely on data and control dependencies among the instructions. For an instance, in case of masking countermeasure, masks are traversed through sensitive idioms having dependency. In other words, the output mask from one idiom is an input mask to another idiom. The compiler forms group of the sensitive idioms having common dependencies using a simplistic program slicing [39] technique. A forward slice can be constructed by traversing forward through the Program dependence Graph (PDG) [9].

A PDG is a directed graph: G=<V, E>. Where

- V is set of vertices containing statements, control predicates.
- E is the set of edges representing flow of data and control through the statements of program.

As a result, this phase of automation reveals the dependencies between idioms and in turn, constructs forward slices for all of the critical data in the program.

## 5 CODE TRANSFORMATION

This is the last phase of proposed compiler. In this phase, compiler applies proper convolution on transformed idioms targeted during the previous step. The output of this phase is secure assembly language program. As usual, the user orders for the protection mechanism via command line arguments

(Ex. method=countermeasure), Where countermeasure is provided in the form of random Precharging or Boolean Masking.

### 5.1 Local Code Transformations

Some convolutions can be applied locally to each sensitive instruction using a peephole optimization [8]. A peephole optimization is one kind of replacement strategy, in which, a block of contiguous instructions is replaced by a more efficient semantically equivalent sequence.

#### 5.1.1 Common Techniques applied in Peephole Optimization [9]:

- Constant folding - Assess constant sub expressions in advance.
  Ex: r1:=3*2 becomes r1:=6

- Strength reduction - Faster Operations will be replaced with slower one.
  Ex: r1=r2*2 becomes r1=r2+r2

- Null sequences – Operations that are ineffective will be removed.
  e.g.: r1=r1+0 and r1=r1*1 are not having any effect, therefore liable to remove.

- Combine Operations - Replacement of the few operations with similar effective single operation.
  e. g.: r2:=r1*2

  Becomes r3:=r1+r1.

  r3:=r2*1

- Algebraic Laws - Simplification and reordering of the instructions using algebraic laws.
  r1:=r2

  Becomes r3:=r2

  r3:=r1.

Returning to our discussion, Cryptography widely used extended instruction set [10] to monitor a small set of custom instructions in a general-purpose processor to accelerate the processing of cryptographic workloads. Figure next (see fig 9) shows the data path required for extended instruction set. Figure 9 graphically represents the functional unit for the AES instruction set extensions (ISE FU) proposed in [10]. *Op1* and *op2* denotes the two 32-bit operands input to the functional unit . The opcode of the instruction initiates the operation in the functional unit, producing the 32 bit value as *result*. This functional unit performs the AES transformations including SubBytes, ShiftRows, and MixColumns, as well as their respective inverses . Using this datapath, one can execute a 128-bit AES encryption in 196 clock cycles as comparison, 1,637 cycles required to perform same work without the extended set instruction data path. But this data path is vulnerable to simple channel attack. So the random precharging methodology can be used as a solution to this problem. It increases the resistance against power analysis by randomly charging the data path before (precharge) and after (postcharge) a critical data is processed. This solution involves careful modification of the cryptographic software, and delivers a moderate amount of increment in security thus offers the protection. Moreover the operations required to perform random precharging differ for each device, depending on its power consumption estimates. For example, random precharging would not offer protection for devices that employ precharged busses.

To proceed with this methodology, the target device applicable is 8-bit AVR microcontroller. Initially some sample experiments are performed to accomplish random precharging. The objective behind this is to find an appropriate idiom that really needs the protection. Then compiler would replace these sensitive idiom.

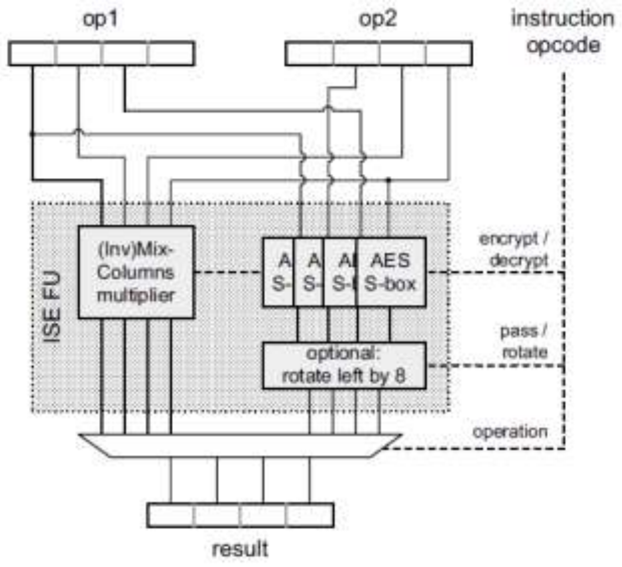


Fig 9: Instruction set extensions for AES as proposed in [10]

Random precharging [10] can also used locally to transform code. example: the data path is randomly charged before and after a critical instruction using randomly generated operands. This approach is effective on devices that have high dynamic power consumption proportional to the Hamming distance between two consecutive cycles' data flowing through a wire, gate, or functional unit; most modern embedded devices exhibit this behavior, since switching activity determines dynamic power consumption. The key idea is to randomize the bits on the critical components, such as a register or data bus; this randomizes the power consumption, since the Hamming distance between a uniformly distributed random variable and a fixed value is also uniformly random.

## 5.2 Global Transformation

Some solutions include information transfer between dependent pair of idioms. In order to do that, the compiler applies global code transformations on the program slices constructed previously. Boolean masking [7] is one of the widely used countermeasures against power analysis attacks.

It requires global code transformation mechanism and provides tight protection during various attacks. The most common form of Boolean masking is x'=x x or r. where:

- x is masked input.
- r is Random Variable.
- x' is masked output

That means, at certain level, Boolean masking performs xor operation on the piece of program with some uniform random values ( we call it as mask). This resultant mask is propagated further as a mask for next level and so on. That means, none of the intermediate values are disclosed during this operation.

### 5.2.1 Actual realization of Boolean Masking [7**]:**

To accomplish a Boolean masking, the data is masked as

x'=x ^ r where r is random variable. To describe it in detail, consider one level computes x3 = x1 ^ x2. Where masked values x1' and x2' could be obtained as x1'=x1 ^ r1 and x2' = x2 ^ r2 respectively in its earlier computation level. Moreover x3' could be realized as x3'= x1' ^ x2' and r3= r1 ^ r2. Based on these, x3 finally obtained as x3=x3' ^ r3.

Returning to the discussion, global transformation includes identification of slice for protection consisting idioms accessing the critical data (we call it as sources). Then compiler operates the Masks on the sources; then traverses the convoluted slice forward along with the masks to intermediate nodes. Finally the masks are removed from the sinks to conclude this process (see fig 10)

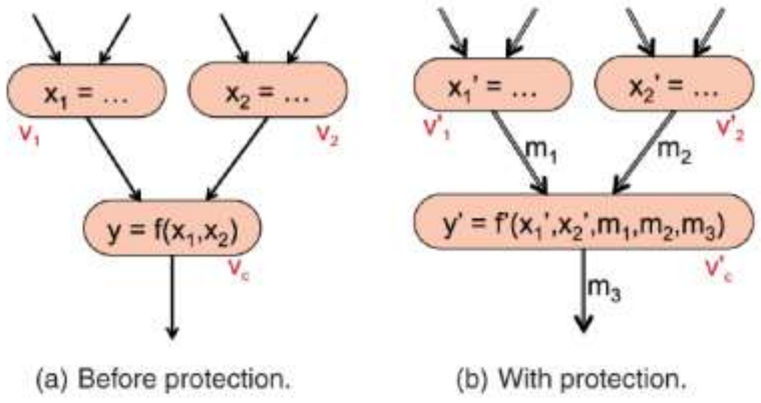


Fig 10: The process of Boolean masking [1]

As shown in figure 10, the convolution is done on sensitive operations with uniformly random values. Further all the intermediate results are also masked without disclosing their behavior. In above hierarchical representation, masks are represented on edges. At last, we require performing unmasking to remove the mask from the sink. As it can be seen, there are two types of operations including linear as well as non linear, implementation of Boolean Masking varies according to their type.

#### 5.2.1.1 Masking linear operations:

Consider node <y= x1 ^ x2 > in the tree shown in Fig. 10.a. The compiler starts the process by masking at the source slice; when it travels further, the masked outputs of earlier step as well as the actual mask values are propagated at next level. According to it, Fig 10.b shows the convoluted slice with powerful protection. Following Algorithm (see fig 11) shows the whole process

Objective: To protects node < y= x1 ^ x2 >

Output:

Vc': masked node.

Inputs:

- m1 : mask propagated from first ancestor of Vc'.
- x1': output of first ancestor of Vc' (i.e. x1'= x1 ^ m )
- m2: mask propagated from first ancestor of Vc'.
- x2':output of second ancestor of Vc'(i.e. x2'= x2 ^ m )

- m3: masks propagated from Vc'.

Return <y'=x1'^x2'^m1^m2^m3>

The confined vertex Vc' contains four operations along with optional load/store operations according to data availability in the registers. The sequence of execution these operations are very important as far as protected output code is considered. To provide maximum amount of security, proposed compiler never discloses the intermediate values ( in example x1,x2 , or y ).

In the protected output, intense care is to be taken in the ordering of two operands for execution.

For ex: < y'=(((x1'^x2')^(m1^m2))^m3)> forms the worst form of ordering.

This form of masking requires availability of registers for the storage of the mask values. Liveness property of register provides this information. If there is a problem of insufficient registers, then some values in the registers are swapped out to the memory and swapped in when there is a no longer need of masks.

### 5.2.1.2 Masking non-linear operations:

According to Shannon's property of confusion, cryptographic algorithms perform non-linear operations to clearly visualize the relationship between plaintext and ciphertext via a substitution box, (S-box).

Non-linear operations in software can be implemented through lookup tables, where the input in the form of plaintext and key, serves as the index for that table. For Ex, the instruction r24=sbox[r24] is implemented as lookup table as the index of sbox is input dependent. Therefore, such types of instructions could be classified as non linear operation.

It is an challenging task to mask a non-linear operations is and it requires to replicate the tables, every time for each mask value.

A load operation is masked as a table lookup by compiler, iff the accessed address is input dependent. The compiler decides whether an address is input dependent by constant propagation, in which certain static analysis is done to verify the queried address is constant or not, if yes, it is not treated as a table lookup otherwise it is supposed to be a table lookup.

Consider the vertex in the tree, representing a sensitive table lookup operation < y=S[x]> and its parent node is producing the value for x. Upon application of mask, x is replaced by x'=x^m.

Now the actual problem starts in computation of y'=y^m3 as we can't use S[x'] in the place of S[x] as it is non-linear operation. So the option is to find out new lookup table Sm' for each new mask value of m. So to create masked output y'=y^m3 using the input x'=x^m1, the applicable formula is, y'=S[x]^m3=S[x'^m1]^m3=S'[m3][x'^m3^m1]. The compiler uses this transformation to protect the output. This transformation is summarized in the following algorithm.

Objective: To Protect the node <y=S[x] >

Output:

Vc' : masked node.

Inputs:

- m1: mask propagated from the only ancestor of
- x1': output of the only ancestor of (i.e., x'=x^m1)
- m3: mask to be propagated from Vc'

Return <y'=S'[m3]^[x'^m3^m1] >

Basically, any operation which is implemented as a table lookup is a non-linear operation and most of cryptographic algorithms are built on top of linear operations or table lookups to simplify the protection.

### 5.3 Optimization of number of masks:

As stated above, a separate mask is required to mask each new non linear operation and obviously produces excess amount of overhead in code-size. To limit these overhead, proposed compiler works on very few number of masks, without compromising in protection.

The compiler follows edge-coloring algorithm to optimize the number of masks. According to Vizing Theory, edge-coloring or graph coloring is an application of colors to the edges of graph such that no two adjacent edges of that graph having the same color. The minimum number of colors required to color the graph is called as *chromatic index* of that graph G, denoted as x'(G).

Based on this theory, the discussion regarding optimum number of masks is expanded.

### 5.4 Unmasking:

Similar to the linear operation, non-linear operations are also require unmasking to remove the mask from the sink finally.

### 5.5 Output code generation:

This is the final step which obtains assembly code from the transformed program representation. This phase is so simple as each node belonging to an instruction or an idiom, can be easily reverted. A code segment generated by standard library calls, is later on inserted by the compiler. Lastly, the intermediate representation is traversed to generate the protected program.

## 6 EXPECTED OUTCOME

In this experiment, two block ciphers are used as benchmarks: AES(Advanced Encryption Standard) and Clefia algorithm. Benchmarks are the suits of tasks required to enhance the performance of application. To simulate AES, AVR controller is used while for Clefia, novel C implementation is used. That's Why, no any hand free assembly language is available. It could be demonstrated from the experimental results, that the compiler can successfully insert the protection mechanisms into an unprotected code. Our target platform is an 8-bit Atmel AVR ATmega microcontroller.

### 6.1 Experimental Setup

Figure 11 shows the experimental setup requirements for the project. An oscilloscope measures power consumption through the differential probe used for digital sampling. A PC examines the power traces and initiates the board to load software and verify correct encryption. The board reduces electronic noise as much as possible as it is designed internally with caliber equipments. . Each experiment will be repeated 25 times and averaged results will be reported further to eliminate random effects caused by measurement.

Power trace are obtained with a 10 ohm resistor associated with microcontroller using the circuit containing digital sampling oscilloscope with differential probe connected to it. The communication between microcontroller and oscilloscope can start and stop measurements. The power traces obtained by the oscilloscope are forwarded to the PC to fight with attacks. The PC runs the cryptographic software on the microcontroller board, and traces the results produced by the microcontroller.

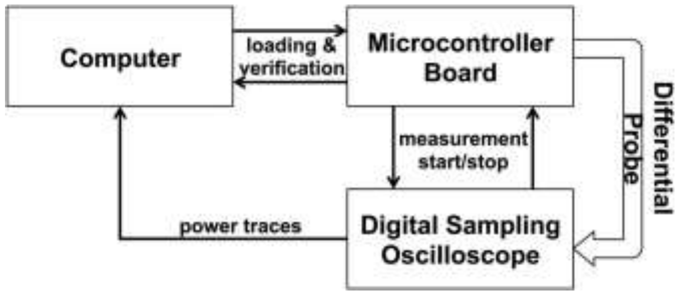


Fig: 11 Experimental arrangement for the project.[1]

### *6.2 Results and Discussion*

Proposed compiler will be able to insert software countermeasures such as Random Precharging and Boolean masking during the presence of attack.

#### 6.2.1 Random Precharging Experiments

In our experiments, command-line parameters including dynamic_analysis_traces.txt which are passed to the compiler: along with threshold=0 and method= randomPrecharging parameter. In order to prepare for that, a file containing power traces, "traces.txt", is required which is randomly-generated by microcontroller using pair of (plaintext, key) pairs, and collecting real-time power measurements. The compiler automatically determined sensitive instructions.

##### 6.2.1.1 Sensitivity Evaluation:

In this subtitle, we have elaborated how the protection mechanism improves security. At first, the compiler applies dynamic analysis to determine the sensitive instructions from cryptographic algorithm. Fig. 12 reports power traces and thereby sensitivity of each clock cycle while the first round of the unprotected AES implementation proceeds. The structure of the AES algorithm yields regular patterns with this kind of analysis. This AES operations is consisting of four main operations :

- AddRound Key (ARK),
- SubBytes (SB),
- Shift Rows (SR)
- MixColumns.(MC).

Internally, AES algorithm is represented as (4 by 4) array, and each operation acts on individual bytes of the state, leading to further regularity.

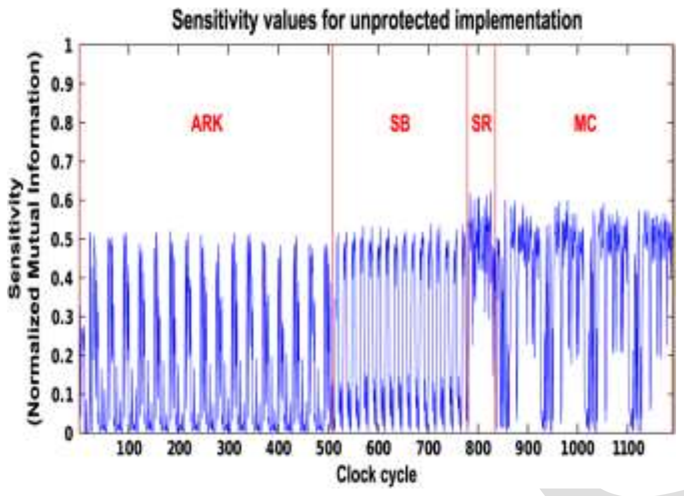


***Fig 12 Sensitivity prediction for each round of AES algorithm [1].***

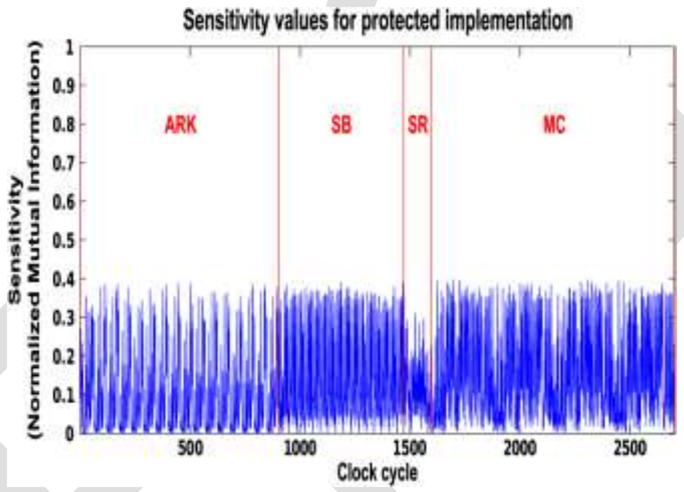


Fig 13: Sensitivity prediction for each round of AES algorithm with random recharging [1].

As shown in fig 13 , Power trace of each clock cycle is traced by random recharging the functional unit of data path before and after the operation.

### 6.2.2 Boolean Masking Experiments

Next, we used our compiler to automatically apply Boolean masking to AES and Clefia. We executed the compiler using command-line parameters static_analysis_critical=Key-Plaintext-method= masking. The parameters inform the compiler to use static analysis (Section3.1) to estimate the sensitivity, to treat the key and plaintext as critical data, and to use Boolean masking as the protection mechanism. The compiler automatically generated protected outputs.

## 7 CONCLUSION

Through this work, advanced proposal for a compiler is published in which its principle requirement is to automatically apply software countermeasures to protect against power analysis attacks .The discussion regarding power analysis attack, side channel attack also enlisted in this work. Software engineers, who do not have any background in cryptography can easily use this compiler. Through an experimental demonstration , it can be elaborated that , the proposed compiler is able to provide security to two principle block ciphers, AES and Clefia. Moreover, this compiler is so intelligent to automatically determine the most sensitive instructions from cryptographic software, and so elegant to protect them with less amount of overhead while obtaining comparable. Therefore such countermeasures are exist in most of today's compiler.

**REFERENCES:**

1) Automatic Application of Power Analysis Countermeasures. by Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne, IEEE TRANSACTIONS ON COMPUTERS, VOL. 64, NO. 2, FEBRUARY 2015.

2) V. Cleemput, B. Coppens, and B. de Sutter ,"Compiler mitigations for time attacks on modern x86 processors", ACMTrans. Archit. Code Optim., vol. 8, no. 4, pp. 1–20, 2012, article no. 23.

a. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne,

3) "A first step towards automatic application of power analysis countermeasures,"in Proc. 48th ACM/EDAC/IEEE Design Autom. Conf. (DAC'11), Jun. 2011, pp. 230–235.

4) M. Barbosa, A. Moss, and D. Page, "Constructive and destructive use of compilers in elliptic curve cryptography," J. Cryptol., vol. 22, no 2, pp 255-281, Apr. 2009.

5) Herbst, E. Oswald, and S. Mangard, "An AES smart card implementation resistant to power analysis attacks," in Proc. Appl. Cryptography Netw. Secur. (ACNS'06), 2006, pp. 239–252.

6) Tiri and I. Verbauwhede,"A digital design flow for secure integrated circuits," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 25, no. 7, pp. 1197–1208, Jul. 2006.

7) Oswald and K. Schramm,"An efficient masking scheme for AES software implementations," in Proc. Int. Workshop Inf. Secur. Appl. (WISA'5), 2005, pp. 292–305.

8) P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in Proc. Adv. Cryptol. (CRYPTO'99), 1999, pp. 398–412.

9) Tip, "A survey of program slicing techniques," J. Program. Languages, vol. 3, no. 3, pp. 121–189, 1995.

10) P. Kocher, "Timing attacks on implementations of Diffie-Hellman RSA, DSS and other systems," in Proc. Adv. Cryptol. (CRYPTO'96), 1996, pp. 104–113.

11) Tip, "A survey of program slicing techniques," J. Program. Languages, vol. 3, no. 3, pp. 121–189, 1995.

12) G. Vizing, "On an estimate of the chromatic class of a P-graph,"Diskret Analiz, vol. 3, pp. 25–30, 1964