ELMO: Evaluating Leaks for the ARM Cortex-M0

Instructions of Use and Documentation

Contents

1 Introduction

This document provides details and instructions on the practical implementation of ELMO and how to use it.

ELMO is a power trace simulator that is able to simulate power traces for any given Thumb binary. The source code is available online at https://github.com/bristol-sca/ELMO. This document accompanies the source code release by providing practical information on how it works and how to use it. For information on the theory and development of the ELMO power model, please consult our paper [5].

Much of what is contained here is provided in the publications [5] and [4] but we here condense this information into a more user friendly format.

This document is divided as follows. First we introduce the basic functionality of ELMO and explain how to simulate power traces. We then go on to detail other features of ELMO that can be used if so desired. These features include the automated fixed vs random leakage detection test, the use of the mask flow methodology and higher order leakage detection and finally the use of ELMO as an energy model. We then provide a guide to the structure of the ELMO program and the different files that are included before explaining how to run the automated ELMO tests and the included examples.

2 Basic ELMO

In this Section we describe the basic working and use of ELMO to simply generate traces. In most cases this may be all that a user requires of ELMO. To improve performance, it is recommended that the additional functionality be removed from ELMO when it is built. To do this, simply remove the defines FIXEDVSRANDOM, MASKFLOW and ENERGYMODEL from the elmodefines.h file.

2.1 Meet ELMO

ELMO is based on a Thumb instruction emulator called Thumbulator [6]. The Thumbulator takes as input a binary program of Thumb assembly, and decodes and executes each instruction sequentially, using a number of inbuilt functions to handle loads and stores to memory and reads and writes to registers. It provides the capability to trace the instruction and memory flow of a program for the purpose of debugging. Our data flow adaptation is built around a linked list data structure: in addition to the instruction type, the values of the two operands and the associated bit-flips from the preceding operands are stored in 32-element binary arrays.

The operand values, and associated bit-flips from the preceding operations, are then used as inputs to the model equations, one for each profiled instruction group. To summarise, simulating the power consumption requires deriving, from the data flow information, the variables corresponding to the terms in the equations: the previous and subsequent instructions, the bits and the bit-flips of each operand, the HW and HDs, and the adjacent bit interactions where relevant (i.e. for `lsls` and `muls`). The variables are then weighted by the appropriate coefficient vector and summed to give a leakage value, which is written to a trace file and saved.

2.2 Non-profiled Instructions

As our model only profiles a subset of the Thumb instruction set, based around the most commonly used instructions for use with symmetric key ciphers, there is a possibility that non-profiled instructions will be encountered in implementations. As we focus on profiling instructions commonly used instructions for use with symmetric key ciphers, we expect that significant cryptographic instructions are likely to be profiled correctly, however we need some way of modelling non-profiled instructions should they be encountered.

As we are profiling individual instructions in sets of three it is important to remember that if one of these instructions in not profiled, the target instruction will be affected. This will be particularly true if the first instruction in the sequence is non-profiled, as the data of the operands of this instruction is used to determine the HD of the target instruction's operands. In considering how to model non-profiled instructions, we therefore want to minimise the impact of the instruction on a triplet but also be clear that we make no claim about the leakage of the instruction. To this end, we model them as ALU (type 1) instructions, as this is the most common instruction type, and set both operands to 0, as we aim to limit the effects of the instruction and make no claims about its leakage. If the first instruction in the sequence is a non-profiled instruction, setting both operands to 0 will simply result in greater leakage of the HW of the operands of the target instruction.

2.3 ELMO Defines

To make configuring ELMO easier for developers, a header file (ELMOdefines.h) is included as part of ELMO that contains a number of options that can be set to change certain parameters in ELMO. A definitive list of these options and parameters along with the default and alternative settings is given in Appendix A however some examples of what can be specified here are whether to produce cycle accurate or instruction accurate traces, the file path and name of output files, the file path and name of a data file to be read from, ELMO progress display information, whether all traces generated will be the same size, the output format of traces (either as binary or printed decimal values) as well as a number of other features. Further options are added with the advances to ELMO given in Section III.

ELMO must be compiled with the desired options having been set in the elmodefines.h file. Although we provide a list of the defines in Appendix A, we note that the addition or modification of these defines to advance or tailor ELMO for a given use is straightforward enough.

2.4 ELMO Functions

We enhance the functionality of ELMO via building a number of functions into ELMO which enable things such as data to be read into and written from the program, a trigger signal to be started and stopped to signal the beginning and end of a trace and the end of a program to be signalled and ELMO stopped. All of these functions are operated through using the `ldr` or `str` instructions with a specific address within the range 0xE0000000 0xE00FFFFF, which on the hardware is assigned as a private peripheral bus region for use for things such as the NVIC, System timer, and System Control Block, the functions of which are either performed by ELMO or are not relevant for the simulation purposes of this work. A complete list of the available functions, their descriptions and how they interact with ELMO is given in Table 6 in Appendix C.

In addition to programming ELMO to carry out the tasks of these functions, we also provide an API that allows developers to easily use these functions when programming in C. The API functions and use is also given in Table 6.

2.5 ELMO Output

The traces can either be output as binary or ASCII. Binary files will require less storage space and allow ELMO to run slightly faster, though they may be less suitable in some instances where a user wishes to easily examine the trace data.

In addition to the trace files, ELMO also outputs a randdata.txt file, which includes any random data that was requested from ELMO via the randbyte function and a printdata.txt file which contains all data printed using the printbyte function.

To assist with diagnostics, ELMO also outputs a full list of the instructions (along with their memory location and machine code representation) in the order in which they are executed. Any activity identified as vulnerable (e.g. by leakage detection tests) can thus be easily tracked back to the original assembly source code. ELMO also outputs the index of any non-profiled instructions encountered in the code to enable developers to more easily take their effects into account.

2.6 Testing ELMO

The correctness of ELMO with regards to the implementation of the methodology outlined in [5] as well as the functional correctness of the emulator (which may have been impacted by our modifications) was tested to ensure the robustness of ELMO.

For the testing of the implementation of the ELMO power model, the model was implemented in Matlab. Every possible sequence of three instructions was then evaluated 1000 times with random data and the corresponding power estimates generated and stored as test vectors. By running the same dynamic compilation program described in [5] with ELMO using the same data as our Matlab test vectors, we are able to generate the same power predictions as the test vectors for all possible sequences of instruction types and compare the results.

The functional correctness of the emulator was tested by comparing a significant number (10,000) of AES ciphertexts, developed using known plaintexts and keys, and comparing these ciphertexts to known test vectors.

These tests were automated in a Python script and intended to be used for basic testing of ELMO following any modifications to the program. Other checks such as the classification of instructions into their correct types were also carried out manually. The automated tests are provided as part of the source code release. See Section 6.4 for more information.

2.7 How to run ELMO

To run ELMO the implementation of the program to be analysed should first be compiled down to a Thumb binary which implements the start and end trigger functions at the beginning and end of the program to be analysed respectively. Each time these functions are encountered a new trace is generated. As many traces as are indicated by the number of start and stop trigger functions will be generated. ELMO will terminate when it encounters the endprogram function. Any other of ELMO's inbuilt user functions that could assist in an implementation could also be used. An example piece of code that generates 200 traces for an implementation of AES with random plaintexts and a fixed key is given below. This kind of trace generation is what could typically be used to perform a DPA attack. This example is similar to that given in Example 1 in the ELMO repository (see Section 6.1).

```
tracenumber = 200;
// Loop sequence for the number of traces to be generated
for(i=0;i<tracenumber;i++){
        // Set inputs and key
        for(j=0;j<16;j++){
                randbyte(&plaintext[j]);
                key[j] = j;
        }
        // Set trigger and call AES function
        starttrigger();
                AES128(ciphertext, plaintext, key);
        endtrigger();
        // Print output
        for(j=0;j<16;j++){
                printbyte(&ciphertext[j]);
        }
    // Terminate program and ELMO
    endprogram();
}
```

In this example, random plaintext bytes are generated using the randbyte function and the key is initialised to 0,1,2...,15 for all 16 key bytes. The trigger is then started, the AES function called and the trigger ended. After this procedure a trace will have been generated and stored in the output folder. Following this, the ciphertext is also printed to a file in case the user wishes to check the operation to ensure it has worked correctly. This procedure will then be carried out 200 times as indicated by the loop. The random data file, generated traces and printed output will be stored in a location specified in the ELMO defines file.

Once this program has been compiled to a Thumb binary file (note the ELMO functions library will need to be linked to use the ELMO functions), it can be run simply by calling the ELMO executable followed by the path to the binary file. If our example program was compiled to a binary file called AES.bin this would be as follows:

./elmo /path_to_binary/AES.bin

Following the running of this command, the traces and data will start to be generated and the progress updated via the Terminal at the specified trace numbers specified in the ELMO defines file. Once the traces have been generated and the endprogram function is called, a summary indicating if the traces were cycle or instruction accurate (specified in the ELMO defines file) and the number of cycles or instructions in the first trace generated. The first trace is used to give the size of the traces assuming the traces are the same size. If the update interval is set to 1 and ELMO set to generate cycle accurate traces, the following output will be seen when our program (which requires 2400 cycles to run) is run:

```
GENERATING TRACES...
TRACE NO: 0000000001
TRACE NO: 0000000002
TRACE NO: 0000000003
TRACE NO: 0000000004
TRACE NO: 0000000005
TRACE NO: 0000000006
...
TRACE NO: 0000000200

SUMMARY:
cycle accurate model
instructions/cycles 2400
```

3 Automated Fixed vs Random

Here we provide details on extensions to the basic functionality provide by ELMO provided in Section 2. One of these extensions is the automation of a genertic leakage detection test, the fixed vs random test [2]. To run this functionality the FIXEDVSRANDOM define must be defined in the elmodefines.h file when ELMO is compiled. Due to the number of traces usually required for this test, it is also recommended that the traces are printed in binary by defining BINARYTRACES and, as the traces should all be the same size, ensuring that the DIFFTRACELENGTH define is excluded to improve performance.

3.1 Generating Traces

When using the fixed vs random functionality of ELMO, the generation of the traces and the selection of the plaintexts, keys and masks (if they are used) are left to the programmer and so need to be programmed into the Thumb binary file. In Chapter 2.4, we describe a list of basic functions (given in Table 6 in Appendix C) that allow a user to interact with ELMO to do things such as start and stop the trigger that can be used to carry out these tasks. In addition to the basic functions provided in Table 6, we also develop a number of other functions that allow a user to interact with the additional functionality provided in this Section. These functions are presented in Table 7 also in Appendix C and referred to according to their API function name in the rest of this Section.

In order to perform a fixed vs random test, traces with a fixed key and fixed plaintext must be generated as well as an equal number of traces with a fixed key and random plain texts. For ELMO to recognise which traces are which, the traces must be generated with the first half of the acquisition being the traces with a fixed key and fixed plaintext (as specified by the programmer) and the second half the traces with a fixed key and random plaintext, where the start and end of the trace is determined by calling the start trigger and end trigger functions respectively. If no masks are used, the fixed traces will be the same for all traces.

As well as being responsible for generating the order of the traces, the developer also has the responsibility of specifying how many traces are to be used in the analysis by starting and stopping the trigger the correct number of times. ELMO automatically takes this number to be half of the total number of times the trigger was started and stopped (with the first half being the fixed traces and the second the random traces). Following the generation of the two blocks of traces, the function to call the fixed vs random test routine should be called.

To make things faster in the event that ELMO stops working midway through the trace generation processes, we also add some other flags that ELMO can be run with to begin taking traces from a specified trace number and also to only run the fixed vs random part of the analysis (in case for some reason ELMO does not complete this part of the analysis and there is no need to retake all the traces).

There are two possible ways of starting the trace generation off from a specified trace number. If the ordering of traces being generated is important (such as if specific data is being input for each trace), 'ghost traces' can be generated. In this case, ELMO runs the program but takes and stores no traces until the number of times the trigger has executed is the number from which the trace taking is set to begin. This has the advantage of ensuring that the data being processed for the desired start trace is correctly aligned with the previous traces, as the program will have executed the previous runs of the program. If the ordering of the data of the traces is not important (for example if random data generated on the fly is being used) it is possibly to simply start the numbering of the traces at the specified number. To only run the fixed vs random analysis, the number of traces sampled for both the fixed and random sets of traces must be specified. The details of these flags is given in Appendix B.

An example of how this code could be written is given in Section 4.4.

3.2 Automated Fixed vs Random

When ELMO encounters the function to call the fixed vs random test routine, it automatically performs a t-test on the fixed and random traces. The results of the fixed vs random test are generated and stored in a file which has each output value of the t-test (corresponding to each instruction) as a line in the file. As ELMO also outputs the instructions that were executed, instructions which are identified as being leaky (by having a t-test value of either > 4.5 or < -4.5, which for large N (> 5000) means the null hypothesis is rejected at the 99.999% confidence interval) can be easily traced to the exact instruction.

4 Mask Flow

This Section explores another extension to ELMO that is designed for use with the masking countermeasure (see [3] for details on masking). To run ELMO with this functionality the MASKFLOW define should be included in the elmodefines.h file when ELMO is built. The FIXEDVSRANDOM define should also be included if the higher order leakage detection tests are to be performed. Due to the considerable performance overhead of using maskflow, it is recommended that it is not included if mask flow is not being used.

4.1 Introducing Mask Flow

Where masking countermeasures are used, evaluating second order leakages against the masks requires preprocessing of the points of the trace which use the same mask. One method of doing this is to multiply the points of the trace which use the mask together [3].

Because one does not normally know which trace points correspond to which masks, one has to exhaustively compute all combinations of d trace points (excluding symmetries). Thus already in the case of a simple first order masking scheme, where only combinations of two points need to be considered, the length of preprocessed traces grows to $(n^2 + n)/2$ (where n is the length of the original traces).

ELMO traces however are different to real world traces in this respect as, although containing the leakage of multiple points in the traces, each leakage point is specifically for a triplet of instructions. This means that we can be certain of the time point of the data dependant power consumption of a single instruction and thus if that instruction is masked, the time point at which any mask dependent power consumption occurs. Another feature is that we are emulating the functionality of the program being evaluated and so we can easily create a data flow model to map the flow of masks through the program and so be able to automatically detect which instructions (and thus corresponding power points) are masked with the same mask.

This means that we can significantly reduce the complexity of carrying out higher order leakage evaluations by using our own mask flow method to map the flow of masks through the program. By automatically detecting how many masks are used in the program and which instructions they are used with we can efficiently perform a fixed vs random test on only the relevant points for all masks that are present, greatly reducing the value of n. This provides a robust and efficient method of higher order leakage detection.

The mask flow method works by modelling each mask as a boolean matrix of $n \times 32$ bits, where 32 is the word size of the ARM Cortex-M0 and n is the number of possible independent mask bits that could mask each bit of the word size. A 1 in the matrix indicates that a specific random bit (as determined by the location on the y-axis) masks the corresponding bit of the word on the x-axis. A 0 means that it does not. One matrix is generated for each operand of an instruction and as each matrix contains all the masking information of each bit of the operand, each unique mask, and it's level of security (as understood through the number of independent random bits masking each bit of the word), can be deduced for each instruction. For the ease of the explanation we specialise our explanation to the case of first-order masking, but it can be readily extended to higher masking orders.

In this way, the mask flow method provides an adapted form of the data flow method described in [1], but works on mask rather than key data. The advantage of representing the mask flow in this way over other data flow methods is that it gives an exact picture of the nature and security (by providing the number of mask bits or shares) of the masks used for each instruction in the program.

To model the flow of masks through the program, a $n \times 32$ bit matrix needs to be generated and stored for each operand of each instruction. This is easily done by including two matrices (one for each of the operands) in ELMO's linked list structure of the data flow model that stores the information of the data being processed by each of the operands and the instruction type of each instruction that is used in simulating the power consumption of the instructions.

This matrix is able to store the mask information for each of the instruction's operands, however if we are to map the flow of the masks through the program, we also need to have a method of mapping the flow of masks through the state of the program. This includes the registers in which the output of operations are stored as well as RAM where data can be written or stored to. This is achieved by generating $n \times 32$ bit matrices for each register in which we store the output mask information. We also generate an n by m matrix, where m is the size of RAM, in which we store the mask information for each bit of the state which is stored to memory. In this way, when data is read from or written to memory, the corresponding mask information for each bit of memory being read from or written to is also operated on in the same way as the data.

Finally, we need to adopt a set of rules which describe how an operation on two masked operands affects the mask of the output. These rules need to allow the masks to be tracked properly through the program in such a way that the output model of the mask, in the form of its matrix, correctly represents mask of the output.

We implement our mask flow analysis for use with boolean masking, where the random masks are added and removed from the state using the exclusive-or operation. We therefore adopt the rules shown in Table 1 to model the different instructions according to their types.

Operation	**Matrix Returned**
Load	Load mask matrix from memory into register.
Store	Store mask matrix in register to memory matrix.
Shift Left	Shift matrix left by value of data shift.
Shift Right	Shift matrix right by value of data shift.
Rotate Right	Rotate matrix right by value of data rotation.
Exclusive-Or	Exclusive-or operation of all corresponding bits of the two operand matricies.
Other Arithmetic Operations	Zero matrix containing no mask information.

Table 1: Rules for mask matrix output for operations on mask matricies.

For memory operations the mask information is simply loaded or stored to the memory matrix, ensuring that mask information is not lost during memory operations. For shift and rotate instructions the mask simply shifts with the data to ensure that the mask reflects the correct bits of the data which are masked. As we are analysing boolean masking, the exclusive-or operation exclusively-ors all bits of the two mask matricies of the operands. This insures that all operations that would lead to the addition, subtraction or changing of a mask are taken into consideration. As we are only considering boolean masking, for simplicity all other arithmetic instructions return a zero mask matrix.

4.2 How to Use the Mask Flow

The first stage of using the mask flow analysis is to initialise the matrix of the memory location of the mask in RAM. This is essential as it introduces the mask into the program. In oder to do this, we developed an additional inbuilt function in ELMO that initialises the mask flow (the initialise mask flow function). This function assumes an 8 bit mask where each bit in the mask is random and independent of all other bits in the mask. This assumption is important for the mask flow analysis as it needs to know whether each bit of the mask applied to each bit of the state is the same random bit used elsewhere or a new random mask bit as this will affect the security level of the mask.

The initialise mask flow function therefore effectively creates a diagonal line of ones in the matrix which is eight bits by eight bits. This indicates that each of the eight bits of the memory location are masked by one bit of an independent mask. In order to specify different random masks, we developed a related ELMO function that specifies the bit number to start from on the y axis of the matrix (the mask flow start function). This allows multiple independent mask bits to mask a single bit of the state by recalling the function but specifying a different start point.

This is shown in Figure 1 and Figure 2. Both of these show the mask matrix of a 32 bit operand with n (the size of the number of possible mask bits) equal to 16. Figure 1 shows 32 bits of memory or a register or operand that is masked with a single 16 bit mask that is used twice so that a single bit of the mask is used on two bits of the 32 bit state. The overall effect here is that each bit is masked with a single bit however the same mask bits are only reused once. To initialise this mask configuration, you could run the initialise mask flow function four times for the memory address of each byte, using the set mask flow start function to change the start point for the initialisation to 8 from 0 for bytes 2 and 4.

```
00000000000000010000000000000001
00000000000000100000000000000010
00000000000001000000000000000100
00000000000010000000000000001000
00000000000100000000000000010000
00000000001000000000000000100000
00000000010000000000000001000000
00000000100000000000000010000000
00000001000000000000000100000000
00000010000000000000001000000000
00000100000000000000010000000000
00001000000000000000100000000000
00010000000000000001000000000000
00100000000000000010000000000000
01000000000000000100000000000000
10000000000000001000000000000000
--------------------------------
11111111111111111111111111111111
```

Figure 1: Single 16 bit mask

```
00000001000000010000000100000001
00000010000000100000001000000010
00000100000001000000010000000100
00001000000010000000100000001000
00010000000100000001000000010000
00100000001000000010000000100000
01000000010000000100000001000000
10000000100000001000000010000000
00000001000000010000000100000001
00000010000000100000001000000010
00000100000001000000010000000100
00001000000010000000100000001000
00010000000100000001000000010000
00100000001000000010000000100000
01000000010000000100000001000000
10000000100000001000000010000000
--------------------------------
22222222222222222222222222222222
```

Figure 2: Two 8 bit mask

Figure 2 shows the state when the 16 bit mask shown in Figure 1 is split into two eight bit masks. Here there are still 16 bits of random mask but, unlike in the other case, the same masks are used four times so that four bits of the state are covered by the same mask bits. The advantage of this method however is that each bit of the state is now covered by two mask bits rather than 1. The result of this is that the secret is divided into three shares, which can provide higher levels of security. This configuration could be created by calling the initialise mask flow function eight times, twice for each byte of the 32 bits with each time having a start point of 0 and 8 respectively.

Once the mask flow has been initialised, traces can start to be taken for the program to be evaluated. For the first trace generated, each instruction stores the two matrices associated with each operand in the linked list structure that stores the data flow and instruction type information. After the first trace has ended (as signalled by the end trigger function), the mask information is then analysed to detect if and where any of the same masks are used in the program, where an instruction is deemed to use the mask if at least one of its operands does. As our trace generating method uses the data of the previous instruction operands to determine its power consumption, the instruction following a masked instruction will also be influenced by its masked data. For this reason we also include the masked instruction's subsequent instruction in the masked instruction index list.

Once this list is compiled and we have the indexes of the masked instructions and their respective masks, we can then use this information to ensure that only instructions affected by the same masks are included in the preprocessed traces for higher order analysis. If masks are detected, this happens automatically after the first order fixed vs random where a fixed vs random test is carried out on the preprocessed traces for each of the masks. If no masks are found, then the analysis ends after the initial first order fixed vs random test.

4.3 Other Uses of Mask Flow Output

As well as being a useful tool in making the preprocessing stage less computationally intensive for higher order leakage evaluation, the output of the mask flow analysis also provides useful information for debugging a masked implementation by providing a list of the mask numbers used in each operand of each instruction. Using this information along with the assembly instructions of the program, a developer is able to see exactly which instruction has used which mask. If first order leakages are detected in a masked implementation, a developer can assess which instruction is leaking and which masks (if any) are being used for the instruction.

In addition to this, ELMO can print the matrix output of each mask found during the mask flow analysis to a file. This allows a developer to see the exact nature of the masks and whether implementation errors have inadvertently changed their configuration to produce errors in the code or render it less secure.

4.4 Examples of Using Fixed vs Random and MaskFlow

We here provide an example of how ELMO works by using it to analyse one round of AES masked with two 8 bit masks (one for the key byte, m_k, and one for the state byte, m_p) that is implemented in Thumb assembly. This example is similar to that given in Example 3 of the ELMO repository (see Section 6.3).

The masking method works by first recomputing the SBox to ensure that when the SBox value is loaded, the correct SBox value and mask is returned for the masked statebyte, before calling the AES round function a single time. The trigger is started and stopped before and after calling the AES round function. The example is similar to that given in Chapter 2.7 however we show how to implement the new mask flow and fixed vs random functionality on a masked implementation of AES.

```
// Set mask flow at memory address of plaintext mask
setmaskflowstart(0);
initialisemaskflow(&m_p);
// Set mask flow at memory address of key mask
setmaskflowstart(8);
initialisemaskflow(&m_k);
// Start traces by taking fixed set of traces first
for(fixed=1;fixed<=0;fixed--){
 for(i=0;i<NOTRACES;i++){
  // Reading data from a file ensures that masks are the same
  // for the fixed and random sets of traces
  readdata(&U);
  readdata(&V);
  for(j=0;j<16;j++){
   readdata(&input[j]);
   if(fixed)
    input[j] = fixedinput[j];
   key[j] = fixedkey[j];
   }
  // Mask the SBox
  initialisemaskedAES(output, input, key);
  starttrigger();
   maskedAES128(output, input, key);
  endtrigger();
 }
 // Reset data file to ensure that same masks are used
 // for random set of traces
 resetdatafile();
}
endprogram();
```

The traces are taken in two sections, the first with the fixed key and plaintext and the second for the fixed key and random plaintext. The masks and random data that is used is stored in the file which is accessed using the read data function which reads data into the program from a given file. The file is reset using the reset datafile function after taking the first section of traces (the fixed traces) to ensure that the masks used for both sets of traces are the same.

The mask information is set for each of the masks using the initialise mask flow function with the addresses of the two masks. The set mask flow start function is also used to identify these two different masks as independent masks that provide an independent random mask bit for each state bit that is masked. In this implementation the start bit of the initialisation was set to 0 for m_p and 8 for m_k.

Once the program has been compiled into a Thumb binary, it can be run by ELMO which immediately begins the process of generating the traces. The mask flow analysis is only carried out after the first trace where the number of masks used and their respective instructions used with is determined and then stored for the later analysis. After the traces have been generated the fixed vs random analysis is carried out when the endprogram function is encountered.

4.4.1 First Order Leakage Detection

Once the traces have been generated, the first order leakage detection test begins and informs us if and leaky instructions have been found. The results are also printed to a file which can be used in conjunction with the assembly program file and mask information file to debug the source code.

In the case of our example we are told that 22 instructions leak via first order leakage. This might seem surprising as we have an implementation that should be masked to protect the implementation against first order attacks, indicating that there is either a bug in the implementation of masking or there are other factors contributing to leakage that the developer did not take in to account. Table 2 shows one of these leaky instructions (shown in red) with the instruction before and after (as ELMO analyses instructions in sequences of three) along with the masks used on each of the operands of the instruction.

Instruction	**Operand 1 Mask**	**Operand 2 Mask**
ldrb	0	2
eors	3	2
ldrb	0	3

Table 2: Leaky instruction triplet with leaky instruction in red.

By examining this sequence we can see that both operands of the `eors` instruction are masked with masks 3 and 2 respectively which, if we take the instruction insolation, should eliminate all leakage from the instruction itself. If however we examine the prior instruction (the `ldrb`), we can see that the mask for operand 2 is the same as that for operand 2 of the `eors` instruction (mask number 2). We could therefore conclude that it is likely that interactions between the HD of these two operands has caused the masks to interact in such a way that the underlying data is leaked. This could then be amended by, for example, inserting a dummy instruction that does not use masked operands between the first `ldrb` and the `eors` instructions, preventing this interaction of the masks and the resulting leakage. In this way the mask flow analysis of ELMO can be used to easily understand and help remove subtle flaws in masking implementations.

4.4.2 Second Order Leakage Detection

If masks are identified, ELMO will proceed to preprocess and evaluate the traces mask by mask. In the case of our example a number of distinct masks are found as the implementation works with a number of representations of the two masks we defined at the beginning of the program. We select mask number 3 to evaluate which represents the single mask byte m_p in our example which masks the plaintext and subsequent state returned from the SBox lookup.

Without the mask flow analysis, the preprocessing stage would require each instruction's power profile to be multiplied by every other instruction's, leading to extremely large traces of size $(n^2 + n)/2$. In our case the size of the trace is 452 and so we would end up with traces of size 102378. However, as our mask flow analysis identifies where the masks are used in the trace, we are able to only carry out the preprocessing on only the relevant masked instructions. For mask number 3, this gives us a trace size of 6441, around 16 times smaller than preprocessing in the naive way. This much reduced size of the the trace to be evaluated makes the second order analysis significantly faster and requires significantly less memory.

Carrying out the fixed vs random test on these preprocessed traces, ELMO informs us of the number of preprocessed points which leak. In our case for mask number 3 this number is 600. The output of the t-test is stored to a file and can then be used along with the assembly output and mask information files to identify the specific instruction pairs and thus the instructions that are leaking. This can be done for all detected masks in the program.

5 Energy Model

We here describe how ELMO can also be used as a data-dependent energy model. To use ELMO in this way, the ENERGYMODEL define should be included with the elmo build. The defines POWERTRACES and CYCLEACCURATE should also be defined to ensure that simulated traces represent accurate real world power and timing values.

As ELMO generates an estimate for the power consumption of each instruction and we know the clock frequency of our profiles, using ELMO to generate an estimate for the energy consumption of a single trace is relatively straight forward. By assuming that our power measurement for each instruction is the same for the whole clock cycle, we can simply multiple the power consumption given by ELMO for it by the time taken for each clock cycle to give an estimate for the energy consumption of each instruction. The time required for each instruction will be $250ns$ (two cycles at $8MHz$) for memory instructions, $125ns$ (one cycle) for all other instructions.

We acknowledge that the assumption that the power consumption value given by ELMO remains the same throughout the clock cycle is a limitation to estimating the exact energy consumption. For the development of ELMO we only examine the power peaks of the clock cycles due to this being where the majority of the data dependent power consumption lies.

This means that as we are assuming the power consumption of each cycle to be equal to the peak power consumption of each cycle, the estimates of the energy consumption values will likely be greater than the real world energy consumption values. However, as the power profile for each cycle is similar, we expect that this assumption will maintain the majority of the proportional differences between the data dependencies within instructions as well as between individual instructions. For the purposes of examining and comparing the energy consumption of different sequences of instructions we believe this will be enough.

By using the method of multiplying the time of a clock cycle by the ELMO predicted power value for each instruction in an implementation, we can get an energy estimate for a single trace. However, we will need to profile the energy of multiple traces each processing different, random (but realistic) data to ensure we get a good average energy consumption for the sequence of instructions being analysed.

We therefore require a certain number of traces to be generated from which to draw the samples for our average energy consumption. In our energy modelling extension to ELMO, this number, and the data being processed, is left to the developer who must ensure that the sample size is adequate. ELMO therefore simply sums all of the power consumption estimates for each of the traces while it is processing them and then simply divides the final sum by the total number of traces to give an overall energy consumption estimate.

If a user desires to use the energy model in conjunction with the fixed vs random test, ELMO will simply select the second set (the random set) of traces to use to estimate the energy consumption. As the number of traces in this instance tends to be quite high, this should be enough to give a reasonably robust estimate of the energy consumption.

The total energy consumption of the implementation is given as the sum of the energy consumption of all instructions and provided in the summary box following the termination of the program. In addition to this, a file is produced which contains the average energy consumption of each instruction to determine the energy consumption of the individual instructions.

6 Running the Examples and Tests

Three examples are given with the release of ELMO. One of these simply simulates traces that could be used in a DPA style attack. The other two perform the fixed vs random test on the traces: one on an implementation of AES from the MBed TLS suite the other on one round of a masked implementation of AES.

To run the examples, the GNU ARM Embedded Toolchain will need to be installed. This can be downloaded from `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm`.

6.1 Example 1: AES DPA Trace Generation

Example found in Examples/DPATraces/MBedAES. To run this example, the defines FIXEDVSRANDOM and MASKFLOW should be excluded from the ELMO build. See Section 2 which further describes the working of this example.

This example generates 200 traces for an implementation of AES taken from the Mbed TLS suite. The inputs are random and the key is fixed to 0,1,2,3,,15. The randdata() function will generate a byte of random data and store it in the input address given. The file where this data is stored is specified in elmodefines.h but by default is in the output folder listed as randdata.txt. The output ciphertexts are also printed in a similar way using the printbyte() function.

The example should be compiled using the make file. Once the binary has been generated this can run by ELMO using the command:

./elmo Examples/DPATraces/MBedAES/MBedAES.bin

This should generate the 200 traces. The inputs (stored in randdata.txt) can be used along with these traces to perform a DPA style attack.

The assembly instructions executed for the program can be found in the folder output/asmtraces and the nonprolifed indexes in the folder output/asmtraces.

6.2 Example 2: AES Fixed vs Random

Example found in Examples/FixedvsRandom/MBedAES. To run this example, the define FIXEDVSRANDOM should be included and MASKFLOW should be excluded from the ELMO build. See Section 3 which further describes the working of this example.

This example is similar to Example 1 though generates two sets of traces (fixed and then random) using the CRI recommended key and inputs. At the end of the example the fixed vs random routine is called when the endprogram() function is encountered. The FIXEDVSRANDOM define must be included when building ELMO for this routine to be called.

The example should be compiled using the make file. Once the binary has been generated this can run by ELMO using the command:

```
./elmo Examples/FixedvsRandom/MBedAES/MBedAES.bin
```

This will generate 10000 traces before performing the fixed vs random test. At the end of the test, the number of instructions or cycles that have failed the test (according to the threshold given in elmodefines.h) will be displayed. A file containing the t-statistic of each instruction or cycle will also be printed in the output folder.

6.3 Example 3: Masked AES Fixed vs Random

Example found in Examples/FixedvsRandom/MaskedAES_R1. To run this example, the defines FIXEDVSRANDOM and MASKFLOW should be included with the ELMO build. See Section 4 which further describes the working of this example.

Here, traces are taken for a single round of masked AES and then the first and second order fixed vs random tests are performed. The mask flow functionality is used to generate the mask flow map to be used with the second order fixed vs random analysis.

The implementation of AES is written in assembly and is masked with two 8 bit masks (one for the key byte, m_k, and one for the state byte, m_p) that is implemented in Thumb assembly. The masking method works by first recomputing the SBox to ensure that when the SBox value is loaded, the correct SBox value and mask is returned for the masked statebyte, before calling the AES round function a single time.

As with Example 2, the traces are taken in two sections, the first with the fixed key and plaintext and the second for the fixed key and random plaintext. The masks and random data that is used is stored in the file which is accessed using the readdata function which reads data into the program from a given file. The file is reset using the reset datafile function after taking the first section of traces (the fixed traces) to ensure that the masks used for both sets of traces are the same.

The mask information is set for each of the masks using the initialise mask flow function with the addresses of the two masks. The set mask flow start function is also used to identify these two different masks as independent masks that provide an independent random mask bit for each state bit that is masked. In this implementation the start bit of the initialisation was set to 0 for m_p and 8 for m_k.

After the traces have been generated the fixed vs random analysis is carried out when the endprogram function is encountered.

The example should be compiled using the make file. Once the binary has been generated this can run by ELMO using the command:

```
./elmo Examples/FixedvsRandom/MaskedAES_R1/MaskedAES_R1.bin
```

This will generate 40000 traces before performing the fixed vs random test. At the end of the test, the number of instructions or cycles that have failed the tests (according to the threshold given in elmodefines.h) will be displayed. Note that a t-test is carried out for every separate mask that is encountered. A file containing the t-statistic of each instruction or cycle for the first order test and also for the second order test of each mask will also be printed in the output folder.

6.4 Running the ELMO Tests

There are three separate ELMO tests: the ELMO power model, functionality and fixed vs random tests. They work by comparing generated ELMO results with test vectors generated using alternative implementations of the ELMO functionality.

All test programs are located in the test folder in ELMO. All three tests are automated and can be run by running the elmotest.py python script. For this to work, ELMO must have been compiled to include the elmodefinestest.h header file rather than the standard elmodefines.h file.

The Thumb binaries that the test script runs with ELMO are found in the folder elmotest-binaries. These have already been compiled to work with the test script. The testvectorprograms contains the programs which were used to generate the test vectors.

References

[1] G. Agosta, A. Barenghi, M. Maggi, and G. Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 81:1–81:6. ACM, 2013.

[2] G. Goodwill, J. J. B. Jun, and P. Rohatgi. A testing methodology for side channel resistance validation. NIST non-invasive attack testing workshop, 2008.

[3] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[4] D. McCann and E. Oswald. Practical evaluation of masking software countermeasures on an iot processor. In *International Verification and Security Conference 2017, Thessaloniki, Greece, July 3-5, 2017. Proceedings*, 2017.

[5] D. McCann, C. Whitnall, and E. Oswald. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. *USENIX Security*, 2017.

[6] D. Welch. Thumbulator. `https://github.com/dwelch67/thumbulator.git/`, 2014.

A ELMO Defines

Table 3: Alternative function defines in ELMOdefines.h.

Default	**Alternative**	**Define**
Don't include fixed vs random test	Include fixed vs random test	FIXEDVSRANDOM
Don't include mask flow analysis	Include mask flow analysis	MASKFLOW
Don't include energy model	Include energy model	ENERGYMODEL
All traces are same length	Traces are different lengths	DIFFTRACELENGTH
ASCII traces	Binary traces	BINARYTRACES
Not mean centred	Mean centred	MEANCENTER
Traces show modelled differential voltage	Traces are converted to power consumption	POWERTRACES
Instruction accurate traces	Cycle accurate traces	CYCLEACCURATE = 1
Print asm file of first trace	Print asm files of all traces	PRINTALLASMTRACES = 1
Print non profiled index file of first trace	Print non profiled index files of all traces	PRINTALLNONPROFILED-TRACES = 1

Table 4: Defines in ELMOdefines.h.

Define	**Value**
FIXEDVSRANDOMFAIL	T-statistic where fixed vs random is deemed to fail (usually 4.5).
PRINTTRACENOINTERVAL	Frequency of ELMO progress information printed to the screen.
RESISTANCE	Fixed parameter used for calculating power from the differential voltage.
SUPPLYVOLTAGE	Fixed parameter used for calculating power from the differential voltage.
CLOCKCYCLETIME	Fixed parameter used for calculating power from the differential voltage.
DATAFILEPATH	File name and path to data file from which data is to be loaded using the readbyte function.
RANDDATAFILE	File name and path to contain random data that is generated using the randdata function.
PRINTOUTPUTFILE	File name and path to data file to contain data to be printed using the printbyte function.
TRACEFOLDER	Folder in which to store generated traces.
NONPROFILEDFOLDER	Folder in which to store non-profiled indexes.
ASMOUTPUTFOLDER	Folder in which to store assembly output files.
TRACEFILE	Name of trace files that are generated.
NONPROFILEDFILE	Name of non-profiled index files that are generated.
ASMOUTPUTFILE	Name of assembly output files that are generated.
MASKFLOWOUTPUTFILE	File name and path of file to contain mask information generated by the mask flow method.
FIXEDVSRANDOMFILE	File name and path of file to contain fixed vs random information generated by the fixed vs random analysis.
ENERGYFILE	File name and path of file to contain modelled energy information.
COEFFSFILE	File name and path in which ELMO model coefficients are stored.

B ELMO Flags

Table 5: ELMO flags.

Flag	**Arguments**	**Function**
-fvr	Number of fixed and random traces.	Performs fixed vs random test on pre-generated traces.
-starttrace	Trace number to start from.	Starts trace generation at specified number.
-startghosttrace	Trace number to start from.	Only stores trace information from number specified.

C ELMO Functions

Table 6: Basic ELMO user functions.

Description	**API Function**	**Operation** (`ldr`/`str`)	**Data Value**	**Address**
Start trace simulation (the trigger).	starttrigger()	`str`	1	0xE0000004
End trace simulation (the trigger).	endtrigger()	`str`	0	0xE0000004
Print single byte located at address *addr* to output file.	printbyte(*addr*)	`str`	byte to print	0xE0000000
Load byte of random (generated using the C random number generator seeded with the time) to memory address *addr*.	randbyte(*addr*)	`ldr`	byte of random data	0xE1000004
Read byte from data file (specified in elmodefines.h) to address *addr*.	readbyte(*addr*)	`ldr`	byte of read data	0xE1000000
Marks end of program and terminates ELMO.	endprogram()	`str`	0	0xF0000000
Resets data file pointer.	resetdatafile()	`str`	0	0xE0000046

Table 7: Advanced ELMO user functions.

Description	**API Function**	**Operation** (`ldr`/`str`)	**Data Value**	**Address**
Initialises the keyflow start to number *startbit* for initialisemaskflow function.	setmaskflowstart(*startbit*)	`str`	Start bit	0xE0000044
Initialises 8 bits of mask at address *add* from point specified by setmaskflowstart function.	initialisemaskflow(*addr*)	`str`	0	0xE0000040
Resets mask flow matrices to 0	resetmaskflow()	`str`	0	0xE0000042