2017 IEEE 35th International Conference on Computer Design

Compiler-Assisted Threshold Implementation Against Power Analysis Attacks

Pei Luo*, Konstantinos Athanasiou†, Liwei Zhang‡, Zhen Hang Jiang*, Yunsi Fei*, A. Adam Ding‡, Thomas Wahl†
{silenceluo@ece., konathan@ccs., zhang.liw@husky., zjiang@ece., yfei@ece., a.ding@, wahl@ccs.}neu.edu
*Electrical & Computer Engineering Department, Northeastern University, Boston, MA 02115 USA
†College of Computer and Information Science, Northeastern University, Boston, MA 02115 USA
‡Department of Mathematics, Northeastern University, Boston, MA 02115 USA

***Abstract*—Side-channel attack utilizes side-channel leakages to extract the secret in crypto systems. Various countermeasures for different algorithms and platforms have been proposed to protect crypto systems against such attacks. Manual countermeasure design requires deep understanding of the target algorithm and implementation, and oftentimes is platform-specific and error-prone. In this paper, we propose the construction of Threshold Implementation (TI), a provably secure countermeasure against power attacks, as an automated compiler pass in the open LLVM (Low Level Virtual Machine) framework. Attack results show that the automatically generated TI designs are secure against power attacks. As our proposed scheme implements the countermeasure at the intermediate representation (IR) level, our method can be applied to any cipher software in any programming language, and the generated implementations can be ported to different platforms and architectures.**

I. Introduction

Side-channel attack has been a realistic threat to various cryptographic systems [1], [2]. It utilizes side-channel leakage, such as power consumption and electromagnetic emanation (EMs), to extract the secret embedded in crypto systems [2]. Various countermeasures have been proposed to protect cryptographic systems against power or EM analysis attacks, falling into two categories: power balancing [3] and computation masking [4], [5]. Existing manual implementations of countermeasures require deep understanding of the cipher and the target implementation, and expertise knowledge of side-channel attacks. What's more, the security evaluation is also ad hoc for such manual implementation, as they are error-prone and there lacks proof or guarantee of security [6]. To address these issues, some efforts have been made towards automated protection design against side-channel attacks [6]–[8].

To automatically implement countermeasures for ciphers in different programming languages and platforms, we argue that the compiler is the suitable stage and tool. We choose the open LLVM (Low Level Virtual Machine) framework as the platform, and implement the construction of protected software as an automated pass at the Intermediate Representation (IR) level [9]. Side-channel resilience is incorporated at the IR level. It is thus language-agnostic, and the generated protected design can be ported to different platforms directly using mature backends.

We choose Threshold Implementation (TI), a provable secure countermeasure against power/EM attacks, as the protection scheme. TI splits the original sensitive variables into multiple shares such that the attacker cannot break the target system unless he has control of all the shares [10], [11]. In this work, we implement an LLVM middle-end pass to generate TI design for a given cipher, and use mature available backend passes to generate binary code for the TI implementation for different hardware platforms. This Compiler Assisted Threshold Implementation (CATI) pass does not require any knowledge of the cipher or the target platform. We take AES and SHA-3 as example ciphers, and automatically generate TI designs for their widely used implementations [12]–[14]. We target a commercial ARM Cortex-M3 processor for the generated TI implementation to evaluate the implementation cost and side-channel resilience.

The rest of this paper is as follows. In Section II, we introduce preliminaries of TI and the LLVM framework. In Section III, details of the proposed CATI pass are presented. In Section IV, we evaluate the generated TI implementations, in terms of both resource overhead and the improvement of side-channel resilience. We conclude this paper in Section V.

II. Preliminaries

A. Threshold Implementation

TI is a kind of side-channel attack countermeasure based on secret sharing and multi-party computation. In TI, a variable $x \in F_2^m$ is split into s additive shares, x_i, with $x = \oplus_{i=1}^{s} x_i$. The vector of s shares of x is denoted as: $\mathbf{x} = (x_1, x_2, \cdots, x_s)$. Thus, the knowledge of up to $s-1$ shares is incomplete and does not reveal information of x. In order to implement a TI for a function, $o = F(x, y, z, \cdots)$, in which $o \in F_2^n$, a set of *shared* functions F_i are required, which each produces an output share o_i and together compute the output of F. There are three properties each shared function should satisfy:

1) **Correctness**: $F(x, y, z) = \oplus_i F_i(\mathbf{x}, \mathbf{y}, \mathbf{z})$ for all $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ and (x, y, z) if $\oplus_{i=1}^{s} x_i = x$, $\oplus_{i=1}^{s} y_i = y$, $\oplus_{i=1}^{s} z_i = z$.
2) **Non-completeness**: every function F_i is independent of at least one share of each variable (x, y, z), such that even if the attacker has full control of this F_i, he cannot get any information about the sensitive variables.
3) **Uniformity**: for all $\mathbf{o} = (o_1, o_2, \cdots, o_s)$ satisfying $\oplus_{i=1}^{s} o_i = o$, the number of tuples $(\mathbf{x}, \mathbf{y}, \mathbf{z}, \cdots) \in \mathcal{F}^{ms}$ for which $F_j(\mathbf{x}, \mathbf{y}, \mathbf{z}, \cdots) = o_j$, $1 \le j \le s$, is equal to $2^{(s-1)(m-n)}$ times the number of $(x, y, z) \in \mathcal{F}^m$ for which $o = F(x, y, z, \cdots)$.

It has been formally proved that if the above properties are preserved for each shared function, the generated implementation is secure against power analysis attacks [10], [11].

B. LLVM

The LLVM framework is a collection of modular and reusable compiler and toolchain technologies. LLVM is com-

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.94

posed of three parts: the front-end translates software implementations in different programming languages into LLVM IRs, the mid-level passes optimize and improve the IRs, and the back-end code generators transfer IRs into binary files for different architectures [9].

LLVM IR is similar to assembly language but target-independent. Various middle-end optimization passes can be designed to modify/optimize the given IR, as does our TI scheme. In this work, we design a middle-end CATI pass in LLVM to transfer the original program IR into its TI implementation. We are using LLVM version 4.0.0 together with Z3 V4.5.0 running on Ubuntu version 12.04.5.

III. CATI - Compiler Assisted Threshold Implementation Pass

We propose to obtain threshold implementations for a given software cipher at the LLVM IR level. First, we present the method to find *semi-TI* solutions (without uniformity guarantee) for a given small operation automatically in Sec. III-A. In Sec. III-B, we give details of achieving uniformity for non-uniform solutions. In Sec. III-C, we show the modifications of a given design to make it suitable for automatic TI construction. Then we show how to use the divide and conquer method to obtain the TI solution for a given cipher efficiently in Sec. III-D.

A. Automated Semi-TI Solutions

Boolean operations can be separated into two categories, linear and nonlinear operations. For each linear operation like Shift and XOR, the TI implementation is straightforward, with the randomness, non-completeness and uniformity of the inputs propagating to their outputs directly. For nonlinear operations, automatic TI generation is less straightforward. In this work, we formulate the automatic TI design problem for nonlinear operations as a Boolean Satisfiability (SAT) problem, and use SAT or SMT (Satisfiability Modulo Theories) solvers to solve the problem.

We denote the original program by P and its TI version by P'. The first step in computing P' from P is to create a parameterized abstract syntax tree (AST) that captures all possible Boolean programs satisfying the TI rules up to a bounded size. We call this parameterized AST a **skeleton**, following the notations in [8]. We use AND operation $c = a\&b$ with three shares (a=$a_1 \oplus a_2 \oplus a_3$, b=$b_1 \oplus b_2 \oplus b_3$, c=$c_1 \oplus c_2 \oplus c_3$) as an example here. A candidate skeleton is shown in Fig. 1.

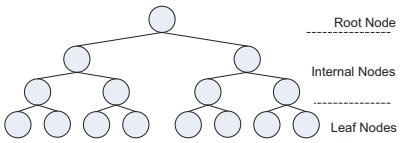


Fig. 1: The structure of candidate tree

The constraints for the ASTs are as follows:

- Each root node is one share of c. For example, three ASTs are used to represent the three shares of c, and the three root nodes are c_1, c_2 and c_3 respectively.
- The internal nodes are instantiated to any bit-level logic operation such as OR, XOR, AND and NOT.
- The leaf nodes are instantiated to shares of the original input variables.

There are several constraints to represent correctness and non-completeness (we ignore uniformity for now):

- The XOR of three root nodes should be equal to c (c=$c_1 \oplus c_2 \oplus c_3$) for any ($a$, b) and (**a**, **b**).
- Each share skeleton cannot include all the shares for one variable in the leaf nodes. For example, the leaf nodes of c_1 skeleton should exclude a_1 and b_1, and similar constraints for c_2 and c_3.

We transfer the above rules into constraints in SMT solvers, and then use SMT solvers to find the TI solutions that satisfy correctness and non-completeness. Initially, there may be no TI solution for an operation with the given skeleton, for example, when the size of skeleton (the depth of the binary tree) is small. We iteratively increase the skeleton size until we find a solution that satisfies the above rules.

As we have ignored uniformity so far in our constraints, the solution returned by the SMT solver will generally not satisfy this requirement, which may incur some weak leakage [10], [11]. To get uniform TI designs, we can either increase the skeleton size or use the strategy of re-masking. Re-masking introduces randomness to add onto the non-uniform outputs.

We use Z3 version 4.5.0 as the SMT solver [15]. An example solution for $c = a\&b$ is given in Fig. 2. In this example, Z3 needs less than one second to find the first non-uniform solution for the given operation. Note that the skeletons of the three shares of c have the same structure, which makes the solving process much more efficient. Skeletons for different shares can have different structures, which gives more possibilities for the TI solutions.

B. Achieving Uniformity with Limited Randomness

For the algorithm in Section III-A, there may be no uniform solution for the given operation with limited skeleton size and number of shares. To achieve uniformity for non-uniform solutions, we can either increase the number of shares or introduce random numbers to re-mask the generated solutions. In this work, we keep the number of shares the same for all operations, and use re-masking to achieve uniformity for certain operations.

It has been shown that outputs of previous cryptographic runs can be re-used as random numbers for re-masking. This will save random number generators (RNGs) in embedded systems [10]. In this work, we use this strategy to save randomness. We check the uniformity of the TI solution, and call corresponding functions to return random numbers stored in memory to re-mask the generated semi-TI if it is not uniform.

C. Minimum Modification of Given Implementation

As stated before, one advantage of the proposed scheme is that it does not require knowing the crypto algorithm and implementation. However, there are two requirements that the program has to conform to so as to ensure the TI properties.

First, in TI, each sensitive variable is split into multiple shares, which must not be recombined in the middle: this violates the rules of non-completeness and will incur leakage. However, some cryptographic implementations may contain secret-dependent control flows. Some modifications are necessary to resolve such branches. Note that, in general, the absence of branch conditions depending on sensitive information is a basic requirement for embedded crypto implementations to avoid timing attacks utilizing imbalanced branches. Thus

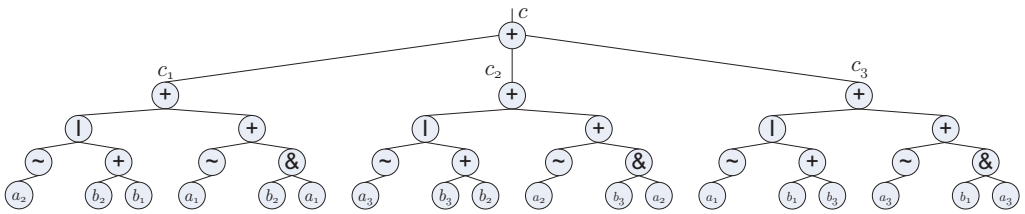


Fig. 2: Example TI solution for the AND operation

the first requirement should be fulfilled even without the consideration of TI design.

Second, for functions in TI, the returned sensitive parameter will also be split into multiple shares. Therefore, the original function should be in *void* type and the sensitive parameter should be passed in and out of the function in a pointer type. To achieve this, designers should modify the target implementation before TI generation.

D. Modular Design

TI implementation is very expensive, and it is beyond the current computation power to find TI solutions for an entire cipher. Instead we rely on the divide-and-conquer method for practical TI construction. We will take modular design, as shown in Fig. 3, and the interfaces between different modules will be important.

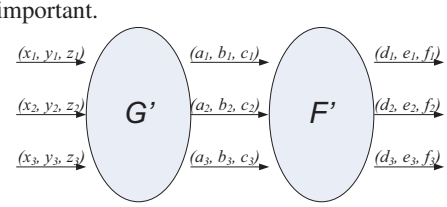


Fig. 3: Divide-and-conquer strategy

We denote the TIs of G and F by $\mathbf{G}'$ and $\mathbf{F}'$ respectively, and the input of $\mathbf{G}'$ by $(\mathbf{x}, \mathbf{y}, \mathbf{z})$, the output of $\mathbf{F}'$ by $(\mathbf{d}, \mathbf{e}, \mathbf{f})$. The output of $\mathbf{G}'$, which is also the input of $\mathbf{F}'$, is denoted by $(\mathbf{a}, \mathbf{b}, \mathbf{c})$. Then the combination of $\mathbf{F}'$ and $\mathbf{G}'$ is the TI of the whole program. This TI design fulfills all the requirements if $\mathbf{G}'$ and $\mathbf{F}'$ fulfill the requirements, respectively.

- **Correctness**: for all possible (x, y, z) and $(\mathbf{x}, \mathbf{y}, \mathbf{z})$, $\mathbf{F}'(\mathbf{G}'(\mathbf{x}, \mathbf{y}, \mathbf{z})) = F(G(x, y, z))$.
- **Uniformity**: if $\mathbf{G}'$ and $\mathbf{F}'$ are uniform respectively, the whole TI module should be uniform.
- **Non-completeness**: in software implementations, there is a synchronization layer (registers) between $\mathbf{G}'$ and $\mathbf{F}'$ to store the intermediate variables $(\mathbf{a}, \mathbf{b}, \mathbf{c})$. For hardware implementations, a synchronization layer of registers should be added to avoid glitches in the output of $\mathbf{G}'$ [16]. Thus non-completeness is ensured.

In this way we separate the original problem into smaller sub-problems and obtain their TIs. In LLVM, we can iteratively scan the IR and combine α LLVM IR instructions together and try to find its TI solution. The size of α will affect the efficiency and resource overhead directly. For larger α, the size of the SAT problem passed to the SMT solver will increase; the solving time for each group of instructions will increase dramatically. For smaller α, a larger number of sub-problems will be generated, which incurs larger resource overhead in the end. Thus there is a balance between the solution speed and area overhead for the proposed method in this work.

We demonstrate the modular TI construction proposed in this work with an example of three instructions. First we modify the function interface according to the description in Sec. III-C, and the shares of $\%x$ are $(\%x.0, \%x.1, \%x.2)$. We process each IR instruction individually, and the TI construction process is shown in Fig. 4.

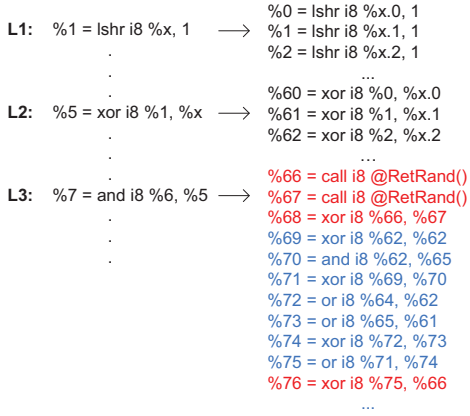


Fig. 4: IR representation of the CATI process

As operations lshr (linear shift right) and XOR are linear, the TI construction of $L1$ and $L2$ are straightforward, and the variable of $\%1$ in the original code becomes three shares $(\%0, \%1, \%2)$ in the generated TI design, while the variable of $\%5$ becomes $(\%60, \%61, \%62)$. For AND operation, the input variables $\%5$ and $\%6$ become multi-share $(\%60, \%61, \%62)$ and $(\%63, \%64, \%65)$, respectively. The computation to generate one share of the output $\%7$ is shown in blue color, with computations for generating other two shares of the output omitted considering the space. As AND operation in $L3$ is non-linear, we use (pseudo) random numbers to re-mask the generated shares, and the re-masking operations are shown in red color. For 3-share AND operation, we use two random numbers $\%66$ and $\%67$ to re-mask the three shares. For example, the first share of $\%7$ is $\%75$ and it is masked with $\%66$ here.

As shown in Fig. 4, the generated TI of AND operation has some redundancies, and this is caused by the structure of skeleton. We can use the optimization passes of LLVM to improve the efficiency, which will be shown in Sec. IV-A.

IV. Result

We automatically generate the 1st-order TI designs for SHA-3 and AES using the proposed scheme. We present the resource overhead and experimental side-channel attack results.

A. Resource Overhead of the Generated TI Implementation

In this paper, we use standard 32-bit implementation of SHA-3 [14] and 8-bit AES implementation based on Canright's S-box [12], [13] as the benchmarks. We make a few modifications to the code according to Section III-C.

We generate the 1st-order TI implementations of both AES and SHA-3 using our algorithm. We also turn on the option '-O3' of LLVM to optimize the generated IR code. Since the pass is at IR level, we use the number of LLVM IR variables to evaluate the overhead in this work. We compare the number of variables for three designs: the original unmasked version, the generated TI design, and the TI design after '-O3' optimization, respectively, and results are presented in Table I.

TABLE I: LLVM IR variables overhead with RNG off

		Original	TI	opt-O3
SHA-3	KeccakF	996	7000	4179
S-box	G4_mul	13	146	100
	G4_sq	5	68	40
	G16_mul	18	152	131
	G256_inv	12	86	69

Table I shows the results for *KeccakF* function in SHA-3 and four functions in AES S-box. The increase in the number of variables for the TI *KeccakF* function is about eight times the original implementation; the '-O3' optimization can reduce 40% variables of the TI implementation. Similarly, for the four S-box functions, the increase for the variables of the TI implementation is about one order of magnitude, and the '-O3' optimization reduces about 30% of the TI implementation.

B. Side-Channel Attacks Results

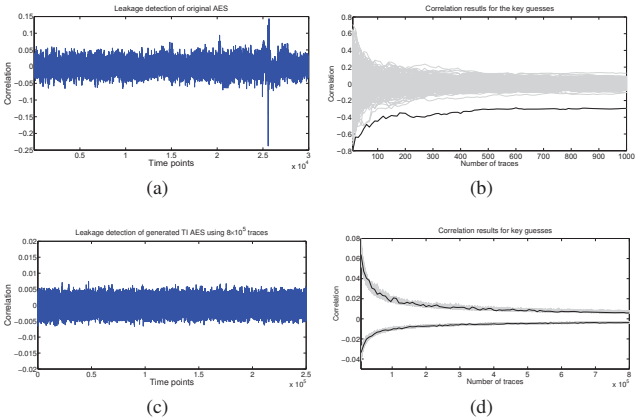
We implement the original and the generated 1st-order TI designs of AES and SHA-3 on a commercial ARM Cortex-M3 (STM32F103C8T6) development board. We sample the EM traces using a Teledyne LeCroy WaveRunner 640zi oscilloscope with the sampling rate of 5G samples/second.

We launched Correlation Power Analysis (CPA) for leakage detection on all the three implementations. For the unprotected AES implementation, the correlation profile for the correct key using 3×10^3 traces is shown in Fig. 5a, where there is strong leakage with the correlation reaching -0.25. The key distinguishing result is shown in Fig. 5b, where the black curve is for the correct key and gray curves are for other wrong key guesses. The correct key stands out clearly among all key guesses with only a few hundred traces.

For the generated TI of AES, the leakage detection results with 8×10^5 traces are shown in Fig. 5c. It shows that the protected AES is secure against first-order attacks. The key guess results are shown in Fig. 5d, in which black curve cannot be distinguished from all the curves (the maximum and the minimum curves are outlined).

V. CONCLUSION

We presented an automated TI design based on LLVM. The proposed scheme is independent of both the programming language and the hardware architecture, and does not require knowledge of the algorithm either. Our proposed method can generate secure TI solutions for a given software implementation automatically. Results show that the generated TI implementations are secure against power (EM) analysis attacks.



(a) (b) (c) (d)

Fig. 5: Side-channel attacks results for original and TI AES

Acknowledgment: This work was supported in part by the National Science Foundation under grants SaTC-1314655, MRI-1337854, and SaTC-1563697.

REFERENCES

[1] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Power analysis attacks of modular exponentiation in smartcards," in *Cryptographic Hardware and Embedded Systems*, 1999, pp. 144–157.

[2] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO*, 1999, pp. 388–397.

[3] K. Tiri, M. Akmal, and I. Verbauwhede, "A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards," in *Prof. European Solid-State Circuits Conf.*, 2002, pp. 403–406.

[4] A. J. Leiserson, M. E. Marson, and M. A. Wachs, "Gate-level masking under a path-based leakage metric," in *Cryptographic Hardware and Embedded Systems*, 2014, pp. 580–597.

[5] M.-L. Akkar and C. Giraud, "An implementation of DES and AES, secure against some attacks," in *Cryptographic Hardware and Embedded Systems*, 2001, vol. 2162, pp. 309–318.

[6] A. G. Bayrak, F. Regazzoni, P. Brisk, F.-X. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proc. Design Automation Conf.*, 2011, pp. 230–235.

[7] G. Agosta, A. Barenghi, and G. Pelosi, "A code morphing methodology to automate power analysis countermeasures," in *Proc. Design Automation Conf.*, 2012, pp. 77–82.

[8] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *Proc. Int. Conf. on Computer Aided Verification*, 2014, pp. 114–130.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp.on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[10] B. Bilgin, "Threshold implementations : as countermeasure against higher-order differential power analysis," Ph.D. dissertation, University of Twente, Enschede, May 2015.

[11] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Int. Conf. Information & Communications Security*, 2006, pp. 529–545.

[12] D. Canright, "A very compact S-Box for AES," *Lecture Notes in Computer Science*, vol. 3659, pp. 441–455, 2005.

[13] "A very compact Rijndael S-box," http://faculty.nps.edu/drcanrig/pub/sboxalg.c.

[14] "Reference and optimized code in C," http://keccak.noekeon.org/KeccakReferenceAndOptimized-3.2.zip.

[15] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[16] T. De Cnudde, O. Reparaz, B. Bilgin, S. Nikova, V. Nikov, and V. Rijmen, "Masking AES with d+1 shares in hardware," in *ACM Proc. Workshop on Theory of Implementation Security*, 2016.