Information Processing Letters 115 (2015) 292–297

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Trace-based schedulability analysis to enhance passive side-channel attack resilience of embedded software

Giovanni Agosta, Alessandro Barenghi *, Gerardo Pelosi, Michele Scandale

Department of Electronics, Information and Bioengineering – DEIB, Politecnico di Milano, Piazza Leonardo da Vinci 32, I-20133 Milano, Italy

ARTICLE INFO

Article history:
Received 18 April 2014
Received in revised form 14 August 2014
Accepted 29 September 2014
Available online 16 October 2014
Communicated by L. Viganò

Keywords:
Compilers
Cryptography
Trace theory
Embedded systems security
Side-channel attacks

ABSTRACT

Side channel attacks (SCAs) are a practical threat to the security of cryptographic implementations. A well known countermeasure against them is to alter the temporal location of instructions among different executions of the code. In this work we provide an algorithm to generate valid schedules of block cipher implementations. The proposed algorithm relies on a trace-theory based analysis and efficiently generates any valid schedule of the implementation under exam, selecting the ones with higher diversity among them. The algorithm was implemented as a pass in the backend of the LLVM compiler suite, and the results of the automated instruction scheduling are provided to validate its effectiveness as an SCA countermeasure employing the whole ISO standard block cipher suite.

1. Introduction

The role of cryptography has grown to a fundamental one in ensuring the security of modern embedded systems. One of the crucial aspects of cryptographic primitives, besides their mathematical security, is to be able to resist the so-called *Side-Channel Attacks* (SCAs). SCAs exploit the fact that several characteristics of an embedded device, such as execution time or instantaneous power consumption, depend on the processed data values [1]. The classic workflow for SCAs aims at recovering the value of the secret key, e.g., of a block cipher, one portion at a time. This is possible since the cryptographic algorithm combines the intermediate data values with a limited number of secret key bits at a time. For instance, employing the power consumption as a side-channel, the first step to perform an SCA is to measure it for the targeted device during a large amount of computations with different input messages. Subsequently, an intermediate operation of the algorithm employing a small portion of the secret-key is selected, and its results are computed for all the possible values of the key portion and input messages. From these hypotheses on the result values of the targeted operation, a series of predictions of the power consumption are made (one for each value of the secret-key portion). Finally, the predicted consumption values are compared with the actual measurements through statistical means to find out which prediction fits best. Such a prediction is the one relying on the correct hypothesis of the value of the secret-key portion.

In this paper, we tackle the security of software implementations of cryptographic primitives, devoting our attention to their protection. We exploit the data dependencies of a cipher implementation to derive different, semantically equivalent, schedules for it. This allows to employ different valid schedules for the cipher at runtime, effectively increasing the difficulty of modeling the execution flow of the cipher. Since the time-alignment of the measurements is a fundamental requirement for a correct SCA [1–3], changing the execution order of the instructions

* Corresponding author. Tel.: +39 02 23993476.
E-mail addresses: agosta@acm.org (G. Agosta), alessandro.barenghi@polimi.it (A. Barenghi), gerardo.pelosi@polimi.it (G. Pelosi), michele.scandale@polimi.it (M. Scandale).

http://dx.doi.org/10.1016/j.ipl.2014.09.030
0020-0190/

of a cipher effectively weakens the effectiveness of the statistical test to infer the key. We propose the first security evaluation of block cipher algorithms in terms of their schedulability properties, analyzing their data dependency graph structures by means of a new and automatic rescheduling technique aimed at maximizing schedule diversity. This provides an effective improvement with respect to the state-of-the-art, which only contemplates some examples of manual, ad-hoc rescheduling of the AES cipher [3], and the insertion of random length delays through dummy instructions [2]. Our automated analysis provides the ground to deploy multiple copies of the block cipher executable code, each one with a different schedule, and randomly pick one of them at each required execution of the cryptographic primitive. We provide a practical validation of our approach analyzing its effects on the whole set of ISO standard block ciphers.

2. Instruction schedulability analysis

In this section we provide the fundamentals of trace theory, the mathematical framework we employ to obtain a schedulability analysis with an acceptable computational effort.

2.1. A formal approach to instruction scheduling

We represent the algorithm through its Control Flow Graph (CFG): a directed graph with the instructions of the algorithm as vertices (nodes) and the control flow dependencies among them as edges. By convention, the CFG has a single node s without any incoming edges, known as the *entry node*, and a single node e without any outgoing edges, known as the *exit node*. Note that a CFG implies a specific schedule of the instructions, i.e., the one imposed by the sequential nature of the original code. Trace theory [4,5] provides an approach to manipulate the schedules of a program, mapping the CFG into a Finite State Automaton (FSA), labeling each CFG node with a letter, and each outgoing edge with the letter of the destination node.

Definition 2.1 *(Program language).* Let Σ be the set of symbols employed for the vertices of the CFG, and Σ^* be the set of all finite sequences of symbols in Σ. Considering the labeled CFG as a FSA, with initial state s, final state e, and transition symbols equal to the letter of the destination state, the *program language* is defined as the subset of Σ^* recognized by the FSA.

The Program Language contains the strings representing all the instruction sequences that result from an execution of the program, i.e., all the legal *program execution logs*. From now on, we will concentrate on a single program execution log and focus on the notion of data dependence within the log instructions to analyze their rescheduling. To model the independence or dependence among instructions, trace theory employs two relations: $\mathcal{I}$ and $\mathcal{D}$, defined as follows.

Definition 2.2 *(Dependence and independence relations).* Let Σ be the program language alphabet. The irreflexive and symmetric relation $\mathcal{I} = \{(a, b) \in \Sigma \times \Sigma \mid a, b \text{ have no data dependence}\}$ is defined to be an *independence* relation. The reflexive, symmetric, and transitive relation $\mathcal{D} = \Sigma \times \Sigma \setminus \mathcal{I}$ is defined to be a *dependence* relation.

Definition 2.3 *(Word equivalence relation).* Let Σ^* be the set of words over the alphabet Σ. The independence relation $\mathcal{I}$ induces an equivalence relation $\sim_{\mathcal{I}}$ over Σ^*. Given two words, $x, y \in \Sigma^*$, $x \sim_{\mathcal{I}} y$ iff there exists a sequence of words $z_0, z_1, \ldots, z_k$ in Σ^* such that $x = z_0$, $y = z_k$ and $\forall i$, $0 \leq i \leq k$ it holds that $z_i = \alpha_i.a_i.b_i.\beta_i$, $z_{i+1} = \alpha_i.b_i.a_i.\beta_i$, $(a_i, b_i) \in \mathcal{I}$, $\alpha_i, \beta_i \in \Sigma^*$ where the notation _._ denotes the concatenation between either letters or words.

Two words in Σ^* are equivalent under $\sim_{\mathcal{I}}$ if and only if the latter can be obtained from the former through a succession of swaps of consecutive letters.

Definition 2.4 *(Trace).* A trace is defined as an equivalence class of the set $\Sigma^*/\sim_{\mathcal{I}}$.

A subset of the set of all possible traces in $\Sigma^*/\sim_{\mathcal{I}}$ is called a *Trace Language*, which is constituted by the representative words of each trace, computed as the outcome of a normalization algorithm. It is thus possible, given a *Program language* and a *Dependence Relation* (resp. *Independence Relation*), to build a *Trace Language*, where each trace contains (at least) one word, i.e., one possible program execution log.

Definition 2.5 *(Dependence graph).* Let Σ be the program language alphabet and $\mathcal{I}$ an independence relation over Σ. Given a word $w = a_0.a_1 \ldots a_{n-1}.a_n \in \Sigma^*$, the dependence graph $\langle w \rangle_{\Sigma,\mathcal{I}}$ of w is defined to be a labeled, directed graph $\langle w \rangle_{\Sigma,\mathcal{I}} = (V, E)$ with each letter $a_i \in w$ bound to a node $v_i \in V$ (thus the number of graph nodes $|V|$ matches the word length $|w|$) and $\forall 0 \leq i, j \leq n$, $v_i, v_j \in V$, $(v_i, v_j) \in E \Leftrightarrow (a_i, a_j) \notin \mathcal{I}$.

Two words w, w' over the alphabet Σ belong to the same trace $T \in \Sigma^*/\sim_{\mathcal{I}}$ iff their dependence graphs are isomorphic: $\langle w \rangle_{\Sigma,\mathcal{I}} \cong \langle w' \rangle_{\Sigma,\mathcal{I}} \Leftrightarrow w \sim_{\mathcal{I}} w'$. The legitimate re-schedules of w can be obtained as the logs corresponding to the graphs which are isomorphic to $\langle w \rangle_{\Sigma,\mathcal{I}}$. In order to find all the legal schedules of a program execution log belonging to a trace T, the most widespread choice is to select the representative of the trace as the one in *Foata Normal Form*.

Definition 2.6 *(Foata Normal Form).* Let Σ be the program language alphabet and $\mathcal{I}$ an independence relation over Σ. A word $w \in \Sigma^*$ is in Foata normal form if either it is the empty word or if it can be decomposed in factors $w = w_0.w_1.\ldots.w_n$ where: (i) each factor w_i is a concatenation of pairwise independent letters, minimal with respect to lexicographical order (ii) given a letter $a \in w_i$ there exists at least one letter b in the consecutive word w_{i+1} (i.e., $b \in w_{i+1}$) such that a and b are dependent, i.e., $(a, b) \notin \mathcal{I}$.

The key point of the Foata normal form is that its factors represent sets of original code instructions that can be