Compiler-based Side Channel Vulnerability Analysis and Optimized Countermeasures Application

Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi
Dipartimento di Elettronica, Informazione e Bioingegneria – DEIB, Politecnico di Milano
Piazza Leonardo da Vinci, 32 – 20133 Milano, Italy
name.surname@polimi.it

ABSTRACT

Modern embedded systems manage sensitive data increasingly often through cryptographic primitives. In this context, side-channel attacks, such as power analysis, represent a concrete threat, regardless of the mathematical strength of a cipher. Evaluating the resistance against power analysis of cryptographic implementations and preventing it, are tasks usually ascribed to the expertise of the system designer. This paper introduces a new security-oriented data-flow analysis assessing the vulnerability level of a cipher with bit-level accuracy. A general and extensible compiler-based tool was implemented to assess the instruction resistance against power-based side-channels. The tool automatically instantiates the essential masking countermeasures, yielding a $\times 2.5$ performance speedup w.r.t. protecting the entire code.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application Based Systems**]: Microprocessor/microcomputer applications;
C.5.3**[Computer System Implementation]**: Microcomputers[portable devices];

General Terms

Security

Keywords

Power Analysis Attacks, Software Countermeasures, Static Analysis

1. INTRODUCTION

Cryptographic primitives are becoming commonplace in embedded hardware security. At the same time, the growing threat of *Side-Channel Attacks* (SCAs) has lead to an increasing interest in designing SCA resistant implementations. SCAs take advantage of the fact that instantaneous power consumption, encryption time, results of artificially induced erroneous computations and/or electromagnetic emissions of an embedded device depend on the processed data and the performed operations. The effectiveness of these techniques in inferring the value of secret parameters, stored in a device in a non-accessible way, justifies the attention towards the implementation choices of HW/SW components as a critical dimension of embedded system design. For instance, the techniques proposed to perform power analysis attacks [8] and their countermeasures have been widely refined in terms of both precision and effectiveness [1]. The classic workflow for power-based SCAs aims at recovering the value of the secret parameter (i.e., the secret key) one portion at a time. This is possible since, during a cryptographic computation, the algorithm combines the secret key bits with the intermediate values involving a limited quantity of them at a time.

DAC'13, May 29 - June 07 2013, Austin, TX, USA.

To perform a power-based SCA, the first step is to measure the power consumption of the targeted device for a large number 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 guessed for all the possible values of the key portion. From these hypotheses on the results, a series of predictions (one for each possible value of the secret key portion) of the power consumption are made. Finally, the predicted consumption values are compared with the actual measured ones through the use of statistical means to find out which prediction fits best. Power-based SCAs affect both hardware and software implementations of cryptographic primitives. Many techniques have been designed to counter these attacks at logic style- or architectural- level, since energy consumption variations are strictly related to them [9]. However, as developing a dedicated hardware solution is expensive, software libraries are commonplace either alone, as a fall-back solution to hardware failures, or to provide complementary functionality with respect to those offered by the hardware platform. These secure software libraries are typically realized in low-level languages such as the target assembly or C language. Even Java-enabled smartcard processors actually expose a hardware support for the execution of Java bytecode, thus making the bytecode their own assembly language. In this work, we address the *protection of software implementations of symmetric ciphers against power analysis*. A very common countermeasure to protect implementations of these ciphers against SCA is to randomize the way sensitive variables are computed through *masking* techniques [6, 9]. The principle is to add one or more random values (*masks*) to every sensitive intermediate variable occurring during the computation. In a masked implementation, each sensitive intermediate value is represented as split in a number of *shares* (containing the randomized sensitive value and the masks employed), which are then separately processed. To this end, the target algorithm is modified to process each share and recombine them at the end of the computation. This technique effectively hinders the attacker from formulating a correct power consumption model, as the instantaneous power consumption is independent from the processed value. Typically, masking techniques are categorized by the number of masks d employed for each sensitive value, which is known as *order* of the masking. A d-th-order masking can always be theoretically broken by a $(d+1)$-th-order attack, i.e. an attack exploiting the combination of $d+1$ measurements of different instructions, during an execution, to build a mask-independent prediction [9, 13, 14]. In practice, the difficulty of carrying out a d-th-order attack increases exponentially with d, due to the difficulty of guessing which time instant is the one when the sensitive computations happen [3]. Even though a high order masking is crucial to ensure good security margins, only a few $d>1$ order masking schemes exist. They impose major performance overheads, and the masking is applied to each cipher with ad-hoc techniques [6, 9, 13, 14].

Contributions

As a major contribution, we provide the definition of a *security-oriented data flow analysis* (SDFA), with forward and backward variants, allowing a precise assessment of vulnerability on every instruction of a low-level intermediate language against power analysis attacks. As a second contribution, we employ the information provided by SDFA to automatically apply a masking protection to the vulnerable portions of the implementation of the target cryptographic primitive. We de-

signed a set of compiler passes, implemented in the widely adopted LLVM compiler framework, parametric with respect to the number of masks to be applied, thus providing a trade-off between performance and protection against higher order attacks, and allowing the use of any language accepted by the LLVM frontend (among which, C). We exploit the fine grain of the information collected by the SDFA to significantly reduce the overhead imposed by the masking techniques, limiting their application only to the instruction of the algorithm which are computationally amenable to power analysis, for a desired security margin. The modular structure of the compiler passes allow the use of the SDFA analysis coupled with other countermeasure approaches. The new compiler passes work on the LLVM intermediate representation, a low-level (RISC-like) representation simple enough to be directly mapped to the actual assembly of most architectures, thus leveraging the compiler infrastructure without the loss of precision associated with high level representations.

Organization of the paper

Section 2 describes the related works. Section 3 summarizes the automated security analysis and countermeasures application as parts of a compiler workflow. Section 4 introduces our security-oriented data flow analysis and Section 5 describes the masking countermeasures and their deployment. Section 6 reports the experimental evaluation, while Section 7 draws our conclusions highlighting future directions.

2. RELATED WORK

The first work tackling the problem of automatically protect software implementations of cryptographic algorithm from power analysis attacks is presented by Bayrak *et al.* in [2]. The authors identify the instructions which are most vulnerable to power analysis running their target implementation and profiling the power consumption of the underlying platform. In this way, they identify the most vulnerable clock cycles of the program execution and associate to each of them the corresponding assembly instruction together with a "sensitivity value". Instructions, whose sensitivity is greater than a chosen threshold are replaced by an appropriate code snippet, which realizes a random pre-charging of either the registers or the memory cells. Note that even if the proposed workflow is general enough, the implemented code transformation step is specific for devices whose power consumption is proportional to the Hamming Distance between two consecutive execution cycles. In this paper, we put forward a novel and fully automated analysis to identify vulnerable instructions. Indeed, the presented technique does not need any profiling information about the power consumption of the target device while it is running the implementation of a cryptographic primitive. The main distinctive feature of our methodology is that it is not affected by completeness and/or accuracy problems bound to the collection of the power measurements, and can be applied even when a development board or cycle accurate simulator is not available, such as in the early stages of development of a processor. More recently, Moss *et al.* [10] proposed a first attempt at automating the process of inserting a 1st order masking scheme in the code of AES using an *ad-hoc* translator. Their scheme relies primarily on *type inference*, a kind of static analysis which is strongly dependent on the source language. To this end, the authors of [10] designed their own Domain Specific Language (DSL) with a specialized type system, which allows type inference. In practice the DSL source code must contain an explicit annotation for variables to be protected (depending on the programmer choices), as there is no automatic evaluation of the security margin bound to each instruction of the program to be executed. In addition, it is worth noting that most encryption primitives, especially for application in embedded systems, are not developed with a DSL, being instead available primarily in C.

3. SECURITY-ORIENTED COMPILER

A modern compiler is a pipeline of analysis and transformation passes processing the program code and transforming its source representation to the desired target representation [11]. The very first and last passes respectively take as input and produce as output the source and target languages. For the representation of all other inputs and outputs, one or more *intermediate representations* (IRs) are used. Analysis passes

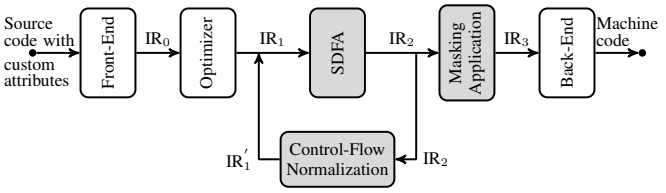

Figure 1: Security-oriented compiler pipeline

compute information from the input IR, and enrich the code through adding either metadata or annotations. Transformation passes, on the other hand, use the information contained in the annotations to modify and optimize the code. IRs have been widely studied in the field of compiler construction to balance the need to keep each analysis or transformation pass as easy as possible, as well as limit the total number of different IRs. A very popular class of IRs is the *Static Single Assignment* (SSA) form, defined as a representation where every variable is defined (i.e., assigned a value) only once [11]. It is important to note that IRs allow also to work at a very low level of abstraction, especially when RISC architectures are involved, as the Back-End passes do not generate vulnerable machine code instructions from the protected IR. This is because register allocation can, at most, spill values to memory, but all sensitive values are already protected, and the instruction selection does not use multiple machine instructions to map a single IR instruction. Figure 1 shows the modified pipeline of our security-aware compiler. The source-language code is extended with custom attributes to allow the developer to provide information to the compiler about the variables containing the "key material" and the "plaintext data" (f.i., the GNU extension mechanism for the C language). The decorated source program is parsed by the *Front-End* to produce the IR, which is optimized using standard optimization passes (f.i., the `-O2` option of either `clang` or `gcc` compiler). The optimized IR is analyzed by a new *Security-oriented Data-Flow Analysis* (SDFA) pass, which adds metadata to each defined variable to identify its level of vulnerability. The SDFA can also identify control flow issues that would prevent a precise analysis and protection of the code. In such a case, the analysis pass invokes a *Control Flow Normalization* pass to transform both loop and conditional structures and to produce a "normalized" version of the input IR, suitable for the computation of vulnerability information in a subsequent execution of the SDFA pass. The vulnerability information is then used by the *Masking Application* pass, which modifies the normalized IR code through applying masking countermeasures where appropriate. The output IR is then translated to the target assembly by the standard *Back-End* pass. The next two sections detail the techniques implemented by the SDFA, Control-Flow Normalization and Masking Application passes.

4. VULNERABLE CODE DETECTION

After lowering the high-level code into an architecture agnostic intermediate representation (IR), it is possible to represent the program as a *control-flow graph* [11] defined as follows:

DEFINITION 4.1 (CONTROL-FLOW GRAPH). *A Control-Flow Graph (CFG) is a directed graph $\mathcal{G}(\mathcal{B}, \mathcal{E})$ with one node $i \in \mathcal{B}$ for each statement of the program ($\texttt{stat}_i$), augmented with two additional nodes i_{in}, i_{out}. An edge $(i, i') \in \mathcal{E}$ is added if the statement $\texttt{stat}_{i'}$ is executed immediately after the statement $\texttt{stat}_i$, and each node has at most two immediate successors. For the first statement ($\texttt{stat}_0$) there is an edge (i_{in}, i_0), while an edge $(j\ ,\ i_{\text{out}})$ is added for each node j bound to a statement ($\texttt{stat}_j$) preceding an exit point of the program.*

In modern production-grade compilers (e.g., LLVM, GCC 4.x), the IR is emitted in "static single assignment" (SSA) form [11]. Indeed, for each non-jump instruction of the IR a fresh virtual register will be used to store the result, thus effectively yielding a one-to-one correspondence between the computed variable values and IR instructions.

To define a *security-oriented data-flow analysis*, aimed at detecting the amount of key material involved in the computation of an interme-

diate value, we will consider a CFG built from an IR in SSA form. The goal of this analysis is to identify a set of nodes of the CFG representing the portions of the program amenable to passive side-channel attacks. The choice of building a single-instruction-per-node CFG is justified by the fact that the application of countermeasures implies a significant performance penalty and should be done sparingly. An instruction is deemed to be vulnerable if computing a model of its behavior for each possible value of the key bits from which its output value depends is computationally feasible. Computing the aforementioned model is the ground on which passive side-channel attacks are built, as its predictions are matched against the measured behavior of the considered device. To trace which computations are affected by some key bits in the program, we need to introduce some terminology related to *data-flow analysis* (DFA). DFA aims at gathering information about the possible set of values calculated at each statement of a program, employing the CFG to determine the propagation paths of each computed value [7, 11]. In our case, the information to be traced is the data-dependence between any bit computed by a program instruction and every bit of the cryptographic key. Such a choice is mandated by the need to consider possible side-channel attack models predicting the behavior of the computation of a single bit within a w-wide value [8]. The aforementioned relation is modeled through an n-bit Boolean lattice $(\mathbb{BV}^n, \sqcup, \sqcap)$, where the elements of the support set

$$\mathbb{BV}^n = \{v_0, \ldots, v_{2^n-1}\} = \{\langle 00\ldots0\rangle, \ldots, \langle 11\ldots1\rangle\}$$

represent all the possible combinations of key bits from which a bit of an intermediate result depends on, thus n equals the key size of the cipher under exam. The bottom of the lattice $\bot$ is represented by the element $\langle 0\ldots0\rangle$, which indicates that no key bits are involved, while the top $\top$ element is $\langle 1\ldots1\rangle$, denoting that all the key bits are involved. The characteristic partial order relations $\succeq$ and $\preceq$ over the lattice elements are defined as follows:

$$\begin{array}{l} v \succeq v' \Leftrightarrow \exists\, v'' \,|\, v' \sqcup v'' = v \\ v \preceq v' \Leftrightarrow \exists\, v'' \,|\, v' \sqcap v'' = v \end{array} \quad v, v', v'' \in \mathbb{BV}^n$$

with the $\sqcup$ operation being defined as the common bitwise inclusive-*or*, and the $\sqcap$ operator being the bitwise *and*.

Our SDFA computes how many key bits are involved in the computation of each intermediate value, i.e., due to the SSA nature of the IR from which the graph is obtained, how many bits are involved in the output of each bit composing the outcome of any instruction $\mathtt{I}$ of the program. To this end, we compute the key propagation for every bit of any size($\mathtt{I}$)-bit wide intermediate result through associating a *leakage vector* $V_{\mathtt{I}}=(v_{\text{size}(\mathtt{I})-1}, \ldots, v_t, \ldots, v_0)$ of size($\mathtt{I}$) elements $v_t \in \mathbb{BV}^n$ to each node of the CFG, which represents a single SSA instruction. Each v_t represents the key bits involved in the computation of the t-th bit of the corresponding intermediate value output by $\mathtt{I}$. We define the "meet" and "join" operations on the leakage vectors (denoted as $\vee$ and $\wedge$, respectively), as the extensions of the aforementioned $\sqcup$ and $\sqcap$. Given two leakage vectors $V_{\mathtt{I}}=(v_{s-1}, \ldots, v_0)$ and $V_{\mathtt{J}}=(v'_{s-1}, \ldots, v'_0)$ of equal size s=size($\mathtt{I}$)=size($\mathtt{J}$), we define the "meet" composition law between $V_{\mathtt{I}}, V_{\mathtt{J}} \in (\mathbb{BV}^n)^{\text{s}}$ as $V_{\mathtt{I}} \vee V_{\mathtt{J}} = (v_{s-1} \sqcup v'_{s-1}, \ldots, v_0 \sqcup v'_0)$. Dually, we define the "join" composition law as: $V_{\mathtt{I}} \wedge V_{\mathtt{J}} = (v_{s-1} \sqcap v'_{s-1}, \ldots, v_0 \sqcap v'_0)$.

Given an instruction, its resistance to passive SCA is formally defined as follows:

DEFINITION 4.2 (INSTRUCTION RESISTANCE). *Consider an IR instruction $\mathtt{I}$ with a size($\mathtt{I}$)-bit output value, and the associated leakage vector $V_{\mathtt{I}}=(v_{\text{size}(\mathtt{I})-1}, \ldots, v_t, \ldots, v_0) \in (\mathbb{BV}^n)^{\text{size}(\mathtt{I})}$. Denoting the Hamming weight of a bit-vector $v_t \in V_{\mathtt{I}}$ as $\text{HW}(v_t)$, the instruction resistance is defined as: $\min_{v_t \in V_{\mathtt{I}} : v_t \neq \bot} \{\text{HW}(v_t), +\infty\}$, that is, the minimum number of key bits influencing a bit of the output value of a sensitive $\mathtt{I}$. An instruction that does not depend on any key bit is conventionally associated to a resistance value equal to ∞.*

To automatically evaluate the resistance of an instruction $\mathtt{I}$, it is necessary to consider the leakage vector associated to each instruction preceding it and take into account which definitions are used by it. This information is captured by the notion of *In-Set* of the instruction. The propagation of resistance information through the specific transformation operated by $\mathtt{I}$ is captured by the notion of *Out-Set* of the instruction.

DEFINITION 4.3 (IN-SET). *Given an instruction $\mathtt{I}$, the input set $in(\mathtt{I})$ is defined as the set of the leakage vectors associated to all the immediate predecessors of $\mathtt{I}$ on the CFG $\mathcal{G}(\mathcal{B}, \mathcal{E})$:*

$$in(\mathtt{I}) \stackrel{\text{def}}{=} \left\{ V_{\mathtt{J}} \mid \mathtt{J} \in \mathcal{B}, \mathtt{J} \in pred(\mathtt{I}), V_{\mathtt{J}} \in (\mathbb{BV}^n)^{\text{size}(\mathtt{J})} \right\}$$

DEFINITION 4.4 (OUT-SET). *Given an instruction $\mathtt{I}$, the output set $out(\mathtt{I})$ is defined as the set of the leakage vectors associated to every immediate predecessor of $\mathtt{I}$ on the CFG $\mathcal{G}(\mathcal{B}, \mathcal{E})$ plus the one of $\mathtt{I}$, $V_{\mathtt{I}} \in (\mathbb{BV}^n)^{\text{size}(\mathtt{I})}$:*

$$out(\mathtt{I}) \stackrel{\text{def}}{=} \{V_{\mathtt{I}}\} \cup \left\{ V_{\mathtt{J}} \mid \mathtt{J} \in \mathcal{B}, \mathtt{J} \in pred(\mathtt{I}), V_{\mathtt{J}} \in (\mathbb{BV}^n)^{\text{size}(\mathtt{J})} \right\}$$

The overall procedure to enact the SDFA is split into two phases: a local analysis and a global one [7]. A local DFA is limited to maximal instruction sequences without jumps (basic blocks) in the control flow. The global DFA works across different basic blocks on the whole CFG.

4.1 Local Security-oriented DFA

DEFINITION 4.5 (LOCAL SECURITY-ORIENTED DFA). *Each instruction $\mathtt{I}$ within a basic block is characterized by an opcode, $op(\mathtt{I})$, an In-set: $in(\mathtt{I})$, and an Out-Set: $out(\mathtt{I})$. The effect of the execution of $\mathtt{I}$ is modelled through a transformation function $\mathcal{F}_{op(\mathtt{I})}(\cdot)$ taking as input its In-set. Therefore, for any instruction $\mathtt{I}$ the following equations can be stated:*

$$\begin{array}{rcl} in(\mathtt{I}) & = & \begin{cases} \emptyset, & \text{if } pred(\mathtt{I})=\emptyset \\ out(\mathtt{J}), & \text{if } pred(\mathtt{I})=\{\mathtt{J}\} \end{cases} \\ out(\mathtt{I}) & = & \mathcal{F}_{op(\mathtt{I})}(in(\mathtt{I})) \end{array}$$

The SDFA solves the set of simultaneous equations derived from the instructions in the basic block through subsequent approximations until a fixed-point is reached. In the particular case of the local SDFA, the convergence is achieved in a single step. The behavior of the transformation function depends on the opcode of the instruction, as the propagation of the key dependencies depends on its nature. Note that, the semantics of each instruction determine also the bit-size (size($\mathtt{I}$)) of its output value. Thus, to compute the corresponding Out-Set it may be necessary to produce a properly sized leakage vector. We denote as $\text{RESIZE}_{\mathtt{I}}(V_{\mathtt{J}})$ the adaptation of the leakage vector $V_{\mathtt{J}}$ to the same size of the instruction $\mathtt{I}$ as follows:

$$\text{RESIZE}_{\mathtt{I}}(V_{\mathtt{J}}) = \begin{cases} (v_{\text{size}(\mathtt{I})-1}, \ldots, v_t, \ldots, v_0), \\ v_t \in V_{\mathtt{J}}, \quad \text{if size}(\mathtt{I}) \leq \text{size}(\mathtt{J}); \\ (\bot_{\text{size}(\mathtt{I})-\text{size}(\mathtt{J})-1}, \ldots, \bot_{\text{size}(\mathtt{J})}, \ldots, v_t, \ldots), \\ v_t \in V_{\mathtt{J}}, \quad \text{if size}(\mathtt{I}) > \text{size}(\mathtt{J}) \end{cases}$$

Let $\text{OPERANDS}(\mathtt{I})$ be the set variables used by $\mathtt{I}$ as operands. As the IR is in SSA form, each variable is defined only once, so, with a small notation abuse, we will use the form $\mathtt{J} \in \text{OPERANDS}(\mathtt{I})$ to denote that instruction $\mathtt{J}$ defines one of the operands of $\mathtt{I}$. For each instruction class of the IR, our analysis assumes the transformation function to be defined as:

$$out(\mathtt{I}) = \mathcal{F}_{op(\mathtt{I})}(in(\mathtt{I})) \stackrel{\text{def}}{=} in(\mathtt{I}) \cup \{V_{\mathtt{I}}\}$$

where the leakage vector of the current instruction $V_{\mathtt{I}}$ is computed according the formulae presented hereafter.

Arithmetic, Bitwise-logic and `cmp` instructions

These instructions can be partitioned in two sets depending on the computation of their leakage vectors.

The first set includes all instructions with an opcode, $op(\mathtt{I})$, specifying a bitwise operation (f.i., `not`, `and`, `or`, `xor`), an `add` or `sub`

operation, with the exception of the `and` and `or` with an immediate operand, as well as `shift`, `zero-` and `sign-extension`. The evaluation of the leakage vector bound either to an `add` or to a `sub` operation is done through considering them as a `xor` operation. This assumption neglects the influence of the carry/borrow propagation in the computation of the result. This is justified by the fact that the most favorable situation for an attacker is when there is no carry propagation (i.e., when the influence of the key bits on each bit of the final outcome is minimized). The computation of the leakage vector of any of the aforementioned instructions is the composition of the leakage vectors in their In-Sets, so that the output bit dependencies (from the key bits) are the ones of the corresponding bits of the input operands added together: $V_{\mathtt{I}}{=}\bigvee_{\mathtt{J}\in\text{OPERANDS}(\mathtt{I})} \text{RESIZE}_{\mathtt{I}}(V_{\mathtt{J}})$. The second set of instructions includes `mul`, `div`, `mod`, and `cmp` operations. Multiplication, division and modulo operations diffuse the information contained in the operand bits, so that every bit of the output depends on every bit of the inputs. Let `I` be any of these instructions, and let $\mathtt{J}\in\text{OPERANDS}(\mathtt{I})$ be the instructions computing the operand values of `I`, with $V_{\mathtt{J}}{=}(v_{\text{size}(\mathtt{J})-1},\ldots,v_t,\ldots,v_0)$ being the corresponding leakage vectors. The leakage vector of the instruction result is computed so that, for each of its bits, the dependencies (from the key bits) of all operands bits are added together: $V_{\mathtt{I}}{=}\bigvee_{\mathtt{J}\in\text{OPERANDS}(\mathtt{I})} \text{RESIZE}_{\mathtt{I}}(\widehat{V}_{\mathtt{J}})$, where $\widehat{V}_{\mathtt{J}} = (\widehat{v}_{\text{size}(\mathtt{J})-1},\ldots,\widehat{v}_t,\ldots,\widehat{v}_0),\ \forall\, t \mid \widehat{v}_t = \bigsqcup_{0\le t<\text{size}(\mathtt{J})} v_t$. Note that, when considering a `cmp` instruction the outcome computed by the instruction is reduced to a single bit.

Bitwise `and` and `or` instructions with an immediate operand
Denoting as imm_i the i-th bit ($0\le i<\text{size}(imm)$) of the immediate operand, we need to define a support leakage vector V_{imm}:

$$V_{imm} = \begin{cases} (\ldots,\langle imm_i,\ldots,imm_i\rangle,\ldots), \\ \qquad \text{with } 0\le i<\text{size}(imm),\ \text{if } op(\mathtt{I}) = \mathtt{and} \\ (\ldots,\langle \neg imm_i,\ldots,\neg imm_i\rangle,\ldots), \\ \qquad \text{with } 0\le i<\text{size}(imm),\ \text{if } op(\mathtt{I}) = \mathtt{or} \end{cases}$$

to model the dependency-cancelling effect of the absorbing elements of bitwise `or` and `and` operations (1 and 0, respectively) on the input leakage vector $V_{\mathtt{J}}$. The output leakage vector $V_{\mathtt{I}}$ is thus obtained removing the cancelled dependencies from the input ones as follows:

$$V_{\mathtt{I}} = \text{RESIZE}_{\mathtt{I}}(V_{imm}) \wedge \bigvee_{\mathtt{J}\in\text{OPERANDS}(\mathtt{I})} \text{RESIZE}_{\mathtt{I}}(V_{\mathtt{J}})$$

`shift` instructions with an immediate operand
Let $\mathtt{J}\in\text{OPERANDS}(\mathtt{I})$ be the instruction producing the non-immediate operand of `I`, and $V_{\mathtt{J}}{=}(v_{\text{size}(\mathtt{J})-1},\ldots,v_0)$ the corresponding leakage vector. The leakage vector associated to `I` is $V_{\mathtt{I}}{=}\text{RESIZE}_{\mathtt{I}}(\widehat{V})$:

$$\widehat{V} = \begin{cases} (v_{\text{size}(\mathtt{J})-1-imm},\ldots,v_0,\bot_{imm},\ldots,\bot_0), \\ \qquad \text{if } op(\mathtt{I}) = \mathtt{shl},\mathtt{ashl} \\ (\bot_{\text{size}(\mathtt{J})-1},\ldots,\bot_{\text{size}(\mathtt{J})-1-imm},v_{\text{size}(\mathtt{J})-1},\ldots,v_{imm}), \\ \qquad \text{if } op(\mathtt{I}) = \mathtt{shr},\mathtt{ashr} \end{cases}$$

The computation of $\widehat{V}$ takes into account the fact that the bits of the output are a permutation of the input ones, possibly discarding some.

Data-dependent `shift` instructions
In this case the non-immediate operands imply considering the outcome of the instruction as an unpredictable result. The corresponding leakage vector is conservatively estimated through removing every dependence from the key bits: $V_{\mathtt{I}}{=}(\bot_{\text{size}(\mathtt{I})-1},\ldots,\bot_0)$.

`store` instruction
`store` operations do not produce any new value. Thus, the following equation applies: $out(\mathtt{I}) = in(\mathtt{I})$ as there is no leakage vector.

`load` instruction
The operands of the `load` instruction can compute an address value that possibly depends on the key bits. Thus, every output bit is considered as dependent on every bit of the address.

Given $\mathtt{J}\in\text{OPERANDS}(\mathtt{load})$, with $V_{\mathtt{J}}{=}(v_{\text{size}(\mathtt{J})-1},\ldots,v_0)$, the information leakage is: $V_{\mathtt{load}}{=}\bigvee_{\mathtt{J}\in\text{OPERANDS}(\mathtt{load})} \text{RESIZE}_{\mathtt{I}}(\widehat{V}_{\mathtt{J}})$, where $\widehat{V}_{\mathtt{J}}{=}(\widehat{v}_{\text{size}(\mathtt{J})-1},\ldots,\widehat{v}_t,\ldots,\widehat{v}_0),\ \forall\, t \mid \widehat{v}_t = \bigsqcup_{0\le t<\text{size}(\mathtt{J})} v_t$. If the address depends on the key and the loaded value also contain some key material the above leakage vector is used as a conservative approximation of the actual one.

`zero-` and `sign-extension` instructions
These two instructions are usually employed in an IR when a change of data type occurs. In this respect, each of them can be managed as an instruction, `I`, with a non-immediate operand that must be extended up to known size. Let $\mathtt{J}\in\text{OPERANDS}(\mathtt{I})$ be the instruction producing the non-immediate operand of `I`, and $V_{\mathtt{J}}{=}(v_{\text{size}(\mathtt{J})-1},\ldots,v_0)$ the corresponding leakage vector. The data associated to the instruction are simply computed as:

$$V_{\mathtt{I}} = \begin{cases} (\bot_{\text{size}(\mathtt{I})-1},\ldots,\bot_{\text{size}(\mathtt{J})},v_{\text{size}(\mathtt{J})-1},\ldots,v_0), \\ \qquad \text{if } op(\mathtt{I}) = \mathtt{zero\text{-}extension} \\ (v_{\text{size}(\mathtt{I})-1},\ldots,v_t,\ldots,v_{\text{size}(\mathtt{J})},v_{\text{size}(\mathtt{J})-1},\ldots,v_0), \\ \text{with } v_t = v_{\text{size}(\mathtt{J})-1},\ \text{where size}(\mathtt{J}) \le t < \text{size}(\mathtt{I}), \\ \qquad \text{if } op(\mathtt{I}) = \mathtt{sign\text{-}extension} \end{cases}$$

4.2 Global Security-oriented DFA and Control Flow Normalization

Given the local SDFA, it is possible to construct a global SDFA through extending the data-flow equations to cover the case where an instruction `I` has multiple immediate predecessors (i.e., $|pred(\mathtt{I})|{>}1$). To this end, we will define the relation between the In-set $in(\mathtt{I})$ of each instruction with multiple predecessors and its Out-set, combining the contribution of the $pred(\mathtt{I})$ through the so-called *confluence* operator. In data-flow analysis techniques, the *confluence* operator is employed to obtain a conservative information regarding the data-flow, as it is not possible to fully predict which value among the ones present in the out-sets of $pred(\mathtt{I})$ will be employed by `I`. This is obtained through preserving only the data-flow information common to all the incoming execution paths, that is, applying a so-called "meet-over-all-paths" policy. In our context, we derive the information associated to the output of an instruction with multiple predecessors through combining them with the "meet" operation on the leakage vectors. More formally our global SDFA is defined as follows.

DEFINITION 4.6 (GLOBAL SECURITY-ORIENTED DFA).
Let $\mathcal{G}(\mathcal{B},\mathcal{E})$ be a control-flow graph and let $\mathtt{I}\in\mathcal{B}$ be an instruction with $|pred(\mathtt{I})|{\ge}1$, then the equations defining its In-set and Out-set are given as follows:

$$\begin{aligned} in(\mathtt{I}) &= \begin{cases} \emptyset, & \text{if } pred(\mathtt{I}){=}\emptyset \\ \bigcup\limits_{\mathtt{H}\in\mathcal{B}} \left\{ \bigwedge\limits_{V_{\mathtt{H}}\in out(\mathtt{J}),\ \mathtt{J}\in pred(\mathtt{I})} V_{\mathtt{H}} \right\}, & \text{otherwise} \end{cases} \\ out(\mathtt{I}) &= \mathcal{F}_{op(\mathtt{I})}(in(\mathtt{I})) \end{aligned}$$

This method would lead to consider a bit of the output of `I` as protected by a key bit, if-and-only-if all the corresponding bits of the outputs of its predecessors are. Although this information is a safe conservative estimate of the instruction resistance, significant enhancements can be obtained through applying some transformations to the IR. We collect such transformations into a Control Flow Normalization pass. Cryptographic algorithms are designed so that the key material diffuses over the cipher state in an increasing fashion at each iteration of the loop (i.e. at each cipher round). Thus, if we detect that the application of the confluence operator ($\wedge$) at the beginning of a loop construct is causing information loss (f.i. when only one predecessor of `I` depends on the key bits), we apply a *loop peeling* transformation to the CFG and restart the analysis. The loop peeling extracts one iteration from the loop block, thus effectively procrastinating the confluence in the control flow and, possibly, allowing the computation of a lossless confluence. This procedure allows to recognize the minimal number of loop iterations to be unrolled. We note that if, at the end of the Global SDFA we detect that the key dependencies do not reach all the live output values (i.e., in this case, the ciphertext), we can automatically point out a cipher design flaw, as this implies that the key is not combined with the whole

cipher state. Another control flow confluence results from conditional statements. In this case we apply a Control Flow Normalization pass, employing a variant of the standard *if-conversion* [11]. If-conversion, to produce efficient code, can only be applied to branches that have a small number of instructions, which is the case of cryptographic algorithms. The common if-conversion transforms a conditional statement either by means of predicated instructions or through computing both branches and employing a `select` instruction to pick the result. We want to transform the construct into a condition free sequence of bitwise logic instructions. For instance, consider the following conditional expression in C language: `res=(r<0) ? r^c : r;`. Denoting as `%r` and `%c` the virtual registers for the variable `r` and `c`, the previous C statement is translated in SSA IR form as shown in the left-pane of the following code snippets:

```
%1=icmp slt i8 %r,0
%2=xor i8 %r,i8 %c
%res=select i1 %1,i8 %2,i8 %r
```

```
%1=ashr i8 %r,7
%2=xor i8 %r,i8 %c
%res=xor i8 %res,%r
```

Note that the comparison with the zero value (checking if `%r` is negative) is substituted with the arithmetic right shift to recover the sign bit of the variable `r`.

4.3 Backward Security-oriented DFA

As it is typical to attack cryptographic algorithms through predicting an intermediate value of the computation preceding the output (the known ciphertext), we will now provide the description of a Backward SDFA to identify the resistance of instructions in this case. Thus, combining the results of the Forward- and Backward-SDFA we will be able to consider all the possible known values the attacker can exploit.

Backward DFAs are constructed similarly to the forward ones, through reversing the relation between In-set and Out-set. However, our SDFA is tailored to the approach used by the attacker to find out portions of the cryptographic key. The general form of the Backward SDFA equations is as follows:

$$out(\mathtt{I}) = \begin{cases} \emptyset, & \text{if } succ(\mathtt{I})=\emptyset \\ \bigcup_{\mathtt{H}\in\mathcal{B}} \left\{ \bigwedge_{V_\mathtt{H}\in in(\mathtt{J}),\, \mathtt{J}\in succ(\mathtt{I})} V_\mathtt{H} \right\}, & \text{otherwise} \end{cases}$$

$$in(\mathtt{I}) = \mathcal{R}_{op(\mathtt{I})}(out(\mathtt{I}))$$

For the arithmetic-logic instructions, the definition of the transformation function $\mathcal{R}_{op(\mathtt{I})}$ is given by: $in(\mathtt{I})=\mathcal{R}_{op(\mathtt{I})}(out(\mathtt{I}))\stackrel{\text{def}}{=}out(\mathtt{I})\cup\{V_\mathtt{I}\}$, where $V_\mathtt{I}=\bigvee_{\mathtt{J}\in \text{USE}(\mathtt{I})} \text{RESIZE}_\mathtt{I}(V_\mathtt{J})$. The main difference from the forward SDFA is the case where one of the operands of `I` contains some key material. Accordingly, the transformation function is modified as follows:

$$in(\mathtt{I}) = \mathcal{R}_{op(\mathtt{I})}(out(\mathtt{I})) \stackrel{\text{def}}{=} out(\mathtt{I}) \cup \{V_\mathtt{I} \vee \text{RESIZE}_\mathtt{I}(V_\mathtt{K})\}$$

where $V_\mathtt{K}=(v_{\text{SIZE(K)}-1},\ldots,v_t,\ldots,v_0)$ is the contribution of each bit of the key (v_t denotes a bit vector that has a single bit set at position t from the least to the most significant). This modification takes into account the fact that an attacker will need to make an hypothesis on the whole key material involved either directly (i.e. as an operand) or indirectly (i.e. in the computation of values depending on the result) with the instruction under exam. Thus, the Backward SDFA has one forward path (the one regarding the use of the key), an uncommon, but not unheard of case for DFA [7]. As the Backward SDFA aims at finding the dependence relations considering the point of view of an attacker in possess of the outputs of the algorithm, the relations among the key bits imposed by the key schedule should be considered in reverse order. To this end, the analysis considers the last values produced by the key schedule as the actual "initial" key and derives the relations with the rest of the key material backwards. The key schedule is identified as the set of instructions that employ, directly or indirectly, only key material and no plaintext. Then, a visit of the whole cipher CFG, starting from the instructions computing the output, is performed to identify instructions that use the key schedule values. The first key schedule values to be found are marked as the "initial" key. The amount of these values which should be considered as "initial", lest the whole key schedule be marked, is bound by their total size being at least the size of the user key, and by them being dependent on all the user key bits. This last condition can be easily checked as a full key schedule forward DFA has been performed at the beginning of this "initial" key identification step. After these "initial" key values are identified, the Backward SDFA computes the dependence from them as imposed by the data-flow equations.

Table 1: Complexity of bitwise masked operations as a function of the masking order d and lookup table size l

Operation	Complexity of masked operation	Ref.
`xor`	$3(d+1)$ `xor`	[6, 13]
`not`	1 `not`	[6]
`and`	$2d(d+1)$ `xor` $+(d+1)^2$ `and`	[6]
`or`	$2d(d+1)$ `xor` $+(d+1)^2$ `and` $+3$ `not`	$a \vee b = \neg((\neg a) \wedge (\neg b))$
table lookup	$2ld$ `xor` $+ld$ `store` $+(ld+1)$ `load`	[14]

5. MASKING COUNTERMEASURES

Masking aims at invalidating the link between the predicted power consumption, associated to an intermediate operation, and the corresponding power measurement. In a masked implementation, each sensitive intermediate value is concealed through splitting it in a number $d+1$ of shares, which are separately processed [6, 9]. The unprotected computation is substituted by three phases: an initial share-splitting, a transformation of the original computation into one processing all the $d+1$ shares and a final recombination, which must yield the same result as the unprotected computation. Table 1 shows the computational costs to mask bitwise operations as a function of the scheme order d. For multi-bit arithmetic operations it is possible to perform conversions between Boolean masked values and arithmetic masked ones and viceversa [5]. The state-of-the-art masking scheme for generic Boolean functions is proven to be secure up to a d-th order attack [6] for any choice of $d\geq 1$. In case the Boolean function is available in the form of a lookup table, the masking of the looked-up values is safe up to the 2nd order according to [4]. The key idea is that, whenever two share-split operands are combined together, fresh random values should be inserted in the computation of the resulting output shares. As the masking countermeasure is particularly computationally demanding (see Table 1) applying it as sparingly as possible, without lowering the security margin of the cipher, is paramount.

Automated Masking Application

We applied the previous masking schemes through our mask application compiler pass, employing the information provided by the SDFA to select which instructions should be protected. Concerning Forward SDFA, we deem an instruction at risk if both its resistance as per Definition 4.2 is lower than a user-selectable threshold, and the computation of its operands depend, directly or indirectly, from the definition of the plaintext. Analogously, in case a Backward SDFA is performed, the instruction is deemed at risk if both the computation of the ciphertext depends on the definition of its output and its resistance is below the aforementioned threshold. The masking application pass (as in Figure 1) visits the CFG in order, and, for each instruction, acts as follows. In case the instruction output needs to be protected, it checks whether the instruction operands are available in masked form or not. In the former case, it simply emits the masked operation corresponding to the unprotected one under exam. In the latter case, it first emits the code required to split the instruction operands into $d+1$ shares, followed by the masked operation code. In case the output of the instruction being visited does not need to be protected, the compiler pass checks whether its operands are masked or not. In case the operands are masked, the pass inserts the share recombination step to obtain an unmasked value. Subsequently, all the uses of the output of the instruction under exam are corrected so that they employ the result of the share recombination.

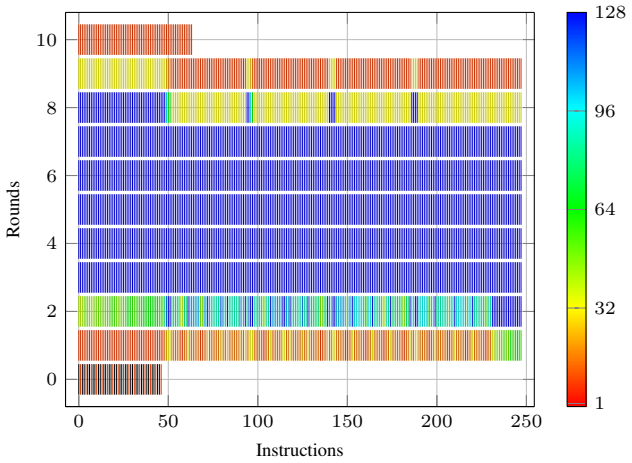

Figure 2: Resistance of sensitive instructions of the AES-128 encryption primitive (`ShiftRow`, `SubByte` as a tabulated S-box, computational `MixColumn`, `AddRoundKey`) in number of key bits protecting the weakest bit of the corresponding outcome. The Key-schedule part is not shown as its instructions are not sensitive

In case the operands of the instruction are already unprotected, the pass continues the visit of the CFG.

6. EXPERIMENTAL EVALUATION

To validate our approach we implemented the SDFA analysis and masking application compiler passes in the current trunk version of the LLVM codebase (Release Candidate in November 2012). We employed two C implementations of the AES cipher, one using the NIST standard 8-bit S-box and the other one with an equivalent optimized computational version. The C code is lowered into the LLVM IR and all the passes for the `-O3` grade optimizations are performed before our SDFA and Masking Application passes are applied. Figure 2 reports the results of running the forward SDFA analysis on AES-128 with tabulated S-box: the initial ADDROUNDKEY (round 0) and 10 rounds of the algorithm are depicted starting from bottom up, with the `x`-axis representing number of instructions for each of them. The results of the same analysis on the computational S-box variant are analogous. The figure shows that, after the end of the second round, no intermediate values need to be protected regardless of the chosen security level as the whole 128 bits of the key influence all the outputs of the operations, thus rendering any SCA as hard as an exhaustive search of the keyspace. Analogous results are obtained from the Backward SDFA, consequentially we can avoid the computationally taxing application of masking techniques to a large part of the AES cipher. The instructions marked in black at the beginning of the cipher are the ones concerning the loading of the plaintext, which have infinite resistance, as the SDFA detects them as not depending on any key bits, but depending on the inputs. By contrast, the end of the cipher does not have any black-marked instruction as the instructions producing the ciphertext are the result of the last key addition: consequentially our backward SDFA correctly deems its result depending on the key bits and marks them with a low resistance, since they should be protected. We chose as a target platform for the benchmarks the "Pandaboard ES development platform". It is endowed with a TI OMAP4460 SoC, containing a dual core Cortex A9-MPCore clocked at 1.2 GHz, 1 MiB L2 cache and 1 GiB DDR2 RAM, running Linaro 12.09 Linux distribution (`armv7l` target). Table 2 reports the results in terms of performance and number of IR instructions of the automatically protected AES implementation obtained through our toolchain, comparing it with the naïve application of the same countermeasure techniques to the whole cipher, as reported [13] and [14]. Both solutions in [14] and in [13] are manually applied and tailored to an 8-bit AES implementation with tabulated and computational version of the S-box, respectively. The timings reported in Table 2 for our solution consider the protection of all the instructions for which the resistance value is lower than 80, a common choice

Table 2: Performance and code size of AES-128 algorithm with both computational and tabulated S-boxes as a function of the number of employed masks, d. All the instructions with resistance lower than 80 are masked. The number of IR instructions includes also the Key-Schedule operations

S-box mode	d	Perf. [μs]	Size [IR-Ins]	Perf. [μs]	Size [IR-Ins]	Speedup
		This work		[13]		
Comp.	Unprot.	60.8	985	60.8	985	–
	1	333.3	5105	870.7	3304	$\times$2.61
	2	983.6	9042	2450.8	6763	$\times$2.49
	3	1983.3	13938	5005.0	11170	$\times$2.52
		This work		[14]		
Tab.	Unprot.	2.2	480	2.2	480	–
	1	94.9	2744	233.9	920	$\times$2.46
	2	132.5	3960	313.7	1660	$\times$2.37

of level of computational effort to deem a cipher secure. Applying the protection to all the instructions with resistance lower than 128 still yields comparable figures (a performance loss lower than 10% with respect to the results in Table 2), without sacrificing any security margin. Our results for the masking applied to the AES variant employing tabulated S-Boxes are $2.46\times - 2.61\times$ faster than their counterpart for a 1-st order masking and $2.37\times - 2.49\times$ for the second order masking with an acceptable instruction count increase due to the partial unrolling of the loops. The code size ratio among IR instruction sizes and the one corresponding to the actually emitted codes (i.e., with a specific ISA) is substantially the same.

7. CONCLUDING REMARKS

We presented a comprehensive security-oriented data-flow analysis and a compiler-based tool to automatically instantiate the essential set of masking countermeasures. Further countermeasure application passes can be easily integrated in the modular structure of the compiler pipeline. Future works will adapt the proposed bit-level security analysis also for HDLs, and extend the automatic application of countermeasures also to hardware circuit designs.

8. REFERENCES

[1] G. Agosta, A. Barenghi, and G. Pelosi. A Code Morphing Methodology to Automate Power Analysis Countermeasures. In *Proc. of the 49th Design Automation Conference*, DAC '12, pages 77–82. ACM, 2012.

[2] 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. of the 48th Design Automation Conference*, DAC '11, pages 230–235. ACM, 2011.

[3] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. J. Wiener, editor, *CRYPTO*, vol. 1666 of *LNCS*, pages 398–412. Springer, 1999.

[4] J.-S. Coron, E. Prouff, and M. Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In P. Paillier and I. Verbauwhede, editors, *CHES*, vol. 4727 of *LNCS*, pages 28–44. Springer, 2007.

[5] B. Debraize. Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking. In Prouff and Schaumont [12], pages 107–121.

[6] Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In D. Boneh, editor, *CRYPTO*, vol. 2729 of *LNCS*, pages 463–481. Springer, 2003.

[7] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[8] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proc. of* CRYPTO'99, pages 388–397. Springer, 1999.

[9] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, 2007.

[10] A. Moss, E. Oswald, D. Page, and M. Tunstall. Compiler Assisted Masking. In Prouff and Schaumont [12], pages 58–75.

[11] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[12] E. Prouff and P. Schaumont, editors. *Cryptographic Hardware and Embedded Systems - CHES 2012. Proc.*, vol. 7428 of *LNCS*. Springer, 2012.

[13] M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In S. M. F.-X. Standaert, editor, *CHES*, vol. 6225 of *LNCS*, pages 413–427. Springer, 2010.

[14] K. Schramm and C. Paar. Higher Order Masking of the AES. In *Proc. of* CT-RSA'06, pages 208–225. Springer, 2006.