Automatic Insertion of DPA Countermeasures

Andrew Moss[1], Elisabeth Oswald[2], Dan Page[2], and Michael Tunstall[2]

[1] School of Computing, Blekinge Institute of Technology,
Karlskrona, Sweden.
andrew.moss@bth.se
[2] Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol BS8 1UB, United Kingdom.
{eoswald, page, tunstall}@cs.bris.ac.uk

Abstract. Differential Power Analysis (DPA) attacks find a statistical correlation between the power consumption of a cryptographic device and intermediate values within the computation. Randomization of intermediate values breaks statistical dependence and thus prevents such attacks. The current state of the art in countermeasures involves manual manipulation of low-level assembly language to insert random masking. This paper introduces an algorithm to automate the process allowing the development of compilers capable of protecting programs against DPA.

Keywords: Differential Power Analysis, Secure Implementations, Compilers.

1 Introduction

Cryptographic software provides a challenging target for software engineering. High-level languages improve programmer productivity by abstracting unnecessary details of the program execution and freeing the programmer to concentrate on the correctness of their implementation. Unfortunately, the details that are generally abstracted away are the behavioral properties of programs, in order to focus on their functional results. In cryptography, the way in which a value is computed may lead to observational differences that an attacker could use to compromise security. If values within the computation that must remain secret, such as cryptographic keys, influence the observational behavior then information will leak and may render the system insecure. If a compiler is allowed to handle the low-level decisions for a given implementation then it must also take how information may leak into account.

State-of-the-art compilers can rival the efforts of a human in producing high performance code. For example, effective methods of register allocation, instruction selection and scheduling often depend on knowledge of the operational details of memory latency and pipeline behavior. Extensions to the execution models targeted by C compilers such as GCC and VisualStudio allow countermeasures to be automatically applied against buffer overflow attacks [1]. A more detailed execution model (e.g. stack frame layout) allows the compiler back-end to perform program transformation that is aware of security constraints.

The increasingly complex threat of physical (e.g. fault and side-channel) attacks on cryptographic implementations offers an interesting extension of the above security case. Automatic resolution of said threat is now an emerging research theme and, alongside more theoretical results in this area (e.g. [2]), a range of concrete compilation systems exist. For example, Molnar et al. [3] construct a binary translation (i.e., compilation) tool that resolves control-flow based leakage using the Program Counter Model (PCM) formalism; Lux and Starostin [4] describe

a tool which detects and eliminates timing side-channels in Java programs (demonsrating the tool by highlighting an attack against the FlexiProvier implementation of IDEA). Likewise, suitable EDA tool-chains [5] can, given some HDL model, automatically implement countermeasures against power-analysis attacks: a back-end which processes some logical netlist can replace standard cells with a secure logic style equivalent (e.g. WDDL [6]) before producing a physical netlist.

Set within this general context, we focus on a specific challenge: given a source program, the goal is to automatically apply masking countermeasures against Differential Power Analysis (DPA) attacks. We make progress toward a general solution, but restrict the domain of source programs to block ciphers; our example case-study focuses on AES.

2 Background

At execution time, a value in a program is a particular bit-pattern. The meaning ascribed to that pattern is dependent upon the context around the code. This basic property of computers makes them flexible, as one pattern of bits can represent many different values depending on the program being executed. However, it can also be a source of error as the meaning of the value is not denoted in the executable code, but rather in the source-level description. Type systems are a method of reducing potential errors by denoting the kind (or type) of value that a particular variable represents. Compilers can then use this type information statically (when compiling the program) to rule out erroneous behavior. In this way types can be seen as a static guarantee of safety Although type theory is a long establish field within the languages and compilers community, the authors believe there is no previous work on using types to describe masking countermeasures.

Static analysis of programs conventionally operates over lattices of values. In this context (and in the remainder of this paper) a lattice is a partially ordered set of values in which any two elements have a uniquely defined infimum and supremum. Operating on lattices of values allows a conservative approximation of program properties: the result is guaranteed to be sound although it may be an over-approximation. Previous work in cryptography has applied analysis over lattices to properties of Information Flow within programs. Classically, Information flow annotates the values in a program as high or low security and prevents high-security information from affecting low-security values to avoid leakage. This classical model was extended [7] to include a principle of *non-interference*. This property can be defined exactly in program semantics by allowing the erasure of the high-security region of a program without any observable difference in the low-security region. Such a definition allows a direct proof that no information has crossed the boundary from one region to the other.

This work shares some similarities to Information Flow; secret values are annotated by the programmer and their secrecy is treated as a value in a lattice allowing the compiler to propagate secrecy information through the program. The main difference is the role of the adversary in the system. In Information Flow the adversary is considered to be on the "edge" of the computation, while execution of code within the high-security region is not observable. In Power Analysis there is possible observation at every point in the program; all secure information must be hidden by masking, but the adversary has the chance to observe all masked operations. While previous work is analytical, a decision is made if a program is secure, this work takes a (potentially) broken program and converts it to a functionally equivalent program that meets the behavioral definition in the model.

Compilers typically operate on an Intermediate Representation (IR) of a program. Input text is parsed into an Abstract Syntax Tree (AST) that represents the structure of program. The AST is then converted to an IR that more closely resembles the execution of instructions

on the target machine. During this process temporary variables are introduced to store the intermediate results in computing expressions. A 3-address form represents each instruction in the program as two input operands, an opcode and a output operand, e.g. $r \leftarrow a$ `xor` b. When the output operand is unique for each instruction this form is called Static Single Assignment (SSA). Multiple write operations to the same variable are renamed to separate instances to ensure this property so that a sequence of the form $x \leftarrow a$ `xor` $b; x \leftarrow x$ `xor` y becomes the sequence $x_1 \leftarrow a$ `xor` $b; x_2 \leftarrow x_1$ `xor` y.

The uniqueness of each target operand implies that loop-free programs form a directed acyclic graph with instructions and variables as vertices, and denoting usage by edges between those vertices. This graph is conventionally termed a *dataflow* representation of the program. In such a graph each vertex v has a set of *ancestors* defined as every vertex where a path exists that reaches v.

3 DPA attacks and mask-based countermeasures

In a DPA attack an attacker tries to recover information about a secret (typically a cryptographic key) by using information about the power consumption of a cryptographic device while it is manipulating the secret in cryptographic operations. To perform such a DPA attack, an attacker selects a so-called intermediate value: e.g. in the specific example that we use to illustrate our work later in this article, the attacker might select the input or output of the AES `SubBytes` operation when applied to the first byte of the AES state.

This intermediate value only depends on a small part of the secret key (in our example only eight bits), which allows an attacker to predict this intermediate value (using knowledge of the input data) for all possible values of that small part of the key. Next an attacker uses a leakage model for the device under attack to map these predicted intermediate values to hypothetical power consumption values: assuming the leakage model is reasonably correct, only the set of hypothetical power consumption values that are related to the correct key guess will match those power consumption values that an attacker can observe from the device itself. Several statistical tools can be used to 'match' hypothetical and real data, e.g. Pearson's correlation coefficient, distance-of-means test, etc.

3.1 Masking to prevent DPA

As can be inferred from the previous description, DPA attacks can only be successful if an attacker can define an intermediate value (based on a suitably small part of the secret key) that is somehow related to the instantaneous power consumption of the device. Thus, DPA attacks can be prevented by making it impossible for an attacker to predict the intermediate values used on the device.

A popular method to this purpose is referred to as 'masking'. Using AES to illustrate the central principle: instead of holding the AES state and AES key 'as they are' one applies a random value to them. For example, the first byte a of the AES state is then represented as pair (a_m, m), with m being the so-called mask which is a number chosen at random from a suitable uniform distribution, such that $a = a_m \oplus m$. Equivalently, the first byte of the first AES round key is then represented as pair (k_n, n), with $k = k_n \oplus n$, and n is chosen at random from a suitable uniform distribution.

In the encryption process itself these masked values need to be processed correctly and securely. For example, if two masked bytes are exclusively-ored, we need to ensure that the result is masked again: $a_m \oplus b_n$ may be carried out but $a_m \oplus b_m$ would result in $a \oplus b$ being

vulnerable to DPA and must not happen. Similarly, table look-ups must be executed such that both inputs and outputs are masked.

Previous work on masking schemes has explored various options for the efficient computation of various cryptographic functions, e.g. the efficient and secure masking of the AES `SubBytes` operation has been extensively discussed. We make use of the work in [8] and [9] by extracting some necessary properties of secure masking schemes. A useful observation made in these previous works was that 'secure against' DPA attacks is synonymous to the concept of statistical independence between variables, i.e. two variables $a_m = a \oplus m$ and a are statistically independent, if the distribution of a_m is independent of the choice of a (for independently chosen uniformly distributed m).

This can be related to some elementary operations involving Boolean variables. Clearly, if a is arbitrary and m is chosen uniformly at random then $a_m = a \oplus m$ is uniformly distributed (and hence its distribution is the same irrespective of the choice of a). Furthermore, $a_{m_a} \times b_{m_b}$, $a_{m_a} \times m_b$, $(a_{m_a})^2$, $p \times a_{m_a}$ (p a constant), and $\sum a_i \oplus m$ can also be shown to be independent of the unmasked values a and b (see [9]). It follows directly that we can guarantee the independence of the output of any operation involving two masked input operands as long as the inputs are independently masked. We note that we have the implicit assumption that only computation leaks, i.e. masks do not contribute to the leakage of the device when only stored in memory).

3.2 Masked variables as a new type

Bringing all this together we can view the process of masking an algorithm as assigning intermediate variables a special 'type' or signature that allows one to infer that they are masked and which mask is being used. This means that rather than holding a variable $a < int >$ that represents, e.g. the first byte of the AES state, we introduce a new type secret such that a secret $< m >$ has the meaning that the variable a is considered to hold private information (and hence must not leak) and is masked by the value m.

Hence, this additional type information allows a compiler to keep track of the 'flow' of masks and intermediate values, to check whether our basic masking rule holds, and if necessary to 'backtrack' variables if there is a problem and add masks to intermediate variables such that the basic masking rule applies. In other words, if a programmer implements a description of an algorithm without any masking, but declares variables related to key and/or state as secret, we can provide the rest of the masking automatically and hence relieve the programmer of that burden.

3.3 Assumptions

We make the following assumptions about the attacker, the programmer, and the device in the remainder of this article.

The attacker has the ability to execute the program repeatedly, and on each execution run an observation (in form of the power consumption) is made on the values computed within the program. The attacker also has access to the inputs and output data of the encryption algorithm, only the masks and keys are hidden from the attacker.

The programmer must mark every value as either secret or public: a valid compilation requires that all secret values have at least one mask. Programmer declared variables and temporary variables inserted during compiler must all meet this requirement.

public values are already known to the attacker. At no stage can a public value (statistically) depend upon a non-public value in a computation. This is partially analogous to Information Flow (in the dependence constraint, also called non-interference).

secret values must be masked with random values which are chosen randomly from a suitable uniform distribution.

The device on which the cryptographic algorithm is implemented supplies (pseudo)random numbers which are uniformly distributed. In each execution run a new set of random numbers is selected and used as masks.

The goal of formalizing a model of countermeasures into a mechanically checkable procedure is not to prove that programs are leakage-free. Although such a goal is desirable with the current state of modelling the complexity of the power consumption characteristics of modern cryptographic devices (e.g. cross-talk in modern technologies which might leak information) it is not tractable. Rather we seek to automate the checking of necessary conditions that must be fulfilled in order for a program to be leakage-free. Although the specific characteristics of a particular device may still cause the program to leak information, the automation of the process enables further study of the specific issues.

4 Algorithm

The algorithm operates directly on an intermediate representation of the program. Our system initially parses the source into an Abstract Syntax Tree (AST), and then converts the AST into a list of instructions in 3-operand form. During conversion all constant bounded loops are statically unrolled and function calls are inlined. The programs that interest an attacker are ciphers with simple control-flow that are converted to straight-line code by this process. The result is a list of instructions and a set of initial variable declarations. Some of the declarations made by the programmer will have security annotations, none of the temporary variables introduced when converting expressions will be annotated.

The algorithm is designed to imitate the process used by a human engineer. The first step is inferring what is known about the security of each value in the program. Our system represents the security annotation as part of the type signature of each variable in the system; the secrecy of a value can be inferred from the secrecy of the operands and the kind of instruction used to create it analogously to the propagation of type information. We refer to this propagation phase as type inference, described in Section 4.1.

After a single type inference pass two outcomes are possible:

1. Inference successfully checked the security of every value in the program and detected no leakages.
2. Inference operated to a point where it detected an error; a type was inferred that showed a leakage of information.

The first case is a successful conclusion and the algorithm terminates by outputing the program in the target assembly syntax. In the second case the algorithm has a record of the particular control point at which leakage occurred. The second phase of the algorithm attempts to repair leakage using a set of program transformations that model the techniques an engineer currently uses in the same situation. The repair phase is described in Section 4.2.

4.1 Type inference

Our prototype used in the experiments operates on a simplified version of the CAO type system [10]. In principle there should be no barriers to implementing the algorithm over the full set of CAO types. The algorithm maintains a security annotation for each type in the system:

```
mask := Wildcard id | Named n
ann := Public | Secret [masks]
```

Every type is either public or secured by a list of masks. Each mask is either named by the programmer or inserted by the compiler. Masks that have been named are used to specify contracts with external pieces of software (i.e. the caller of the routine). Wildcard masks are removed when possible by the compiler. The removal is via substitution of another mask and the process is guarded by the condition that no value can be reduced from more than zero masks to zero masks. In the example these annotations are attached to the follow types:

```
type := byte ann | vector n ann | map ann ann
```

Individual byte variables have their own annotation (and hence set of masks), while vectors are assumed to be masked by the same set. Maps describe functions in which the input and output can be masked separately and are used to denote lookup tables such as S-boxes. Our conversion from AST to 3-address form unrolls loops statically, inlines function calls and convert to an SSA form. As each variable has only a single definition, the program type inference operates in a single forward pass in which the annotation of each variable is inferred from the operation in the instruction and the previously computed annotations of the source operands. The cases for type inference can thus be defined as rules that produce the type on the left when the pattern on the right matches:

public	$\leftarrow$ public `xor` public
secret $\overline{x}$	$\leftarrow$ secret $\overline{x}$ `xor` public
secret $\overline{y}$	$\leftarrow$ public `xor` secret $\overline{y}$
secret $(\overline{x} \cup \overline{y}) \setminus (\overline{x} \cap \overline{y})$	$\leftarrow$ secret $\overline{x}$ `xor` secret $\overline{y}$

These rules can be verified from the definition that a value k with an annotation of secret $\overline{x} = \{x_1, \ldots, x_n\}$ is defined as $k \oplus x_1 \oplus \ldots x_n$, and the same mask in both source operands will cancel under two applications of `xor`. If any annotation is computed to be secret $\emptyset$ then the inference stops and an error is generated at that control point.

The rules for load and store operations are simpler as they rely on the masks being exactly the set of value defined in the vector type being accessed:

secret $\overline{x} \leftarrow$ `load` secret $\overline{x}$ public

The second operand is the index (offset) in memory. After the loop unfolding during conversion these values are constant and thus known to the attacker. The case for a map is slightly more general:

secret $\overline{y} \leftarrow$ `load` secret $\overline{x}$ public

This assumes that the map has type secret $\overline{x} \rightarrow \overline{y}$. A consequence of these minimal definitions is that any case not included as a valid rule will cause the inference to fail with an error. This is commonly referred to as a 'closed world assumption'.

4.2 Repair heuristics

Each of our repair rules is designed to function generally on any supplied input program. However, the set of rules is certainly not complete and requires expansion based on the study of other test cases. As a result of this incompleteness, we will refer to these rules as heuristics, although we emphasize that each rule is sound and guaranteed to improve the security of the program being rewritten.

The repair phase operates after an inference pass, and results in a drastic reduction in the number of cases to consider. The inference pass handles all forward propagation of information:

each temporary value initially has no inferred type and can, therefore, be set to the result of applying the inference rules. As the program is in SSA form this propagation pushes information through the use-def chains in the program. The SSA form is defined implicitly in terms of the definition and uses of values. This removes potential errors from the forward pass and only leaves a few cases where the target type is already fixed:

Weak stores occur when a secret value is stored in a public vector. The algorithm forms the repair by introducing a new copy of the vector protected by a fresh wildcard mask.

Weak maps occur when a secret value is used as an index in a public map (e.g. if a key derived value indexes an S-box).

Mask collisions occur during a store operation when the mask set for the source operand does not equal the mask set for the target vector.

Revelations occur when an instruction with secret operands produces a public value.

Both of the first two cases occur because the annotation of the structure in memory is weaker than the accessing value. In the case of the vector, a new copy is synthesized in which the elements are covered under a fresh wildcard mask. In the case where the map is a random shuffle applied to create a secure copy. The shuffle is defined by an input and an output mask: $S_{m \to n}[i] = S[i \texttt{ xor } m] \texttt{ xor } n$. In both cases substitution is used to convert the program to the secure form: for every following instruction both read and write accesses to the insecure structure are rewritten to use the secure version. A shuffling operation is synthesized to copy the public version into the secret version and inserted directly before the instruction causing the error.

Both of the second two cases occur because the propagation of the mask sets according to the rules defined in the preceding section have yielded a value that is insecure. In these cases the problem cannot be fixed where it is observed and the algorithm must find a source for the error that can be fixed. For each operand in the error-causing instruction the algorithm considers the set of ancestor values. For each ancestor the algorithm examines the effects of flipping a single mask at a time in the mask-set of the ancestor. These single mask flips correspond to the effect of inserting one `xor` instruction on the ancestor value and rewriting the subsequent parts of the chain to use the altered value. In each case the algorithm checks if the problematic value is fixed, and whether any other values are revealed. If no successful repairs are found, the algorithm then considers pairs of flips amongst the ancestor values, triples etc. When the set of successful flips is non-empty the algorithm uses the number of inserted flips as a simple metric to choose the least-cost solution.

4.3 Combined process

The two phases described are executed in alternating order.

1. Infer the types of all values starting from programmer declarations.
2. If an error occurred then perform a repair action on the program.
3. Repeat until no errors are found or a repair cannot be performed.

5 Worked example

The `MixColumns` stage of the AES block cipher has been studied extensively in the context of power analysis and the application of countermeasures. The input language for our prototype compiler is derived from CAO. The rich type system of CAO is especially suitable for analysis [10] and previous work has shown that the collection types are of benefit in compiling block ciphers [11]. For the `MixColumns` stage our prototype requires a small number of data-types and so the input language is a subset of CAO.

Algorithm 1 The Automatic Masking Algorithm

procedure repairWeakMap(pos,inst)
  m = new Wildcard
  n_T = secret $\{m\} \rightarrow o_T$ **where** origMap = $i_T \rightarrow o_T$
  substitute every use of original map with $mapM$ from pos onwards
  insert instructions at pos to compute $mapM[i] := orig[i$ `xor` $o_T]$ `xor` m
procedure repairWeakStore
  m = new Wildcard
  rewrite vector type in *declarations* to secret $\{m\}$
procedure repairAncs
  $ancs := \{anc \mid anc \in \text{UDC}(operand), operand \in inst\}$
  $worklist := 2^{ancs}$ (sorted in increasing size and computed lazily)
  for each *ancset* in *worklist* **do**
    Choose one mask in each ancestor in *ancset*
    Flip the mask in each maskset and rerun the inference
    if no new values are made insecure and the problem value is made secure **then**
      Append (*mask*, *ancset*) to *results*
    end if
  end for
  Sort *results* by size of *ancset*
  if length *results* > 0 **then**
    Insert flip operations into program
  else
    Abort with an error
  end if
procedure topLevel
  while not finished **do**
    bindings := *declarations*
    for each $inst, pos$ in *prog* **do**
      extract r, a, b, *operation* from *inst*
      a_T, b_T := lookup a, b in *bindings*
      r_T := infer from *operation*, a_T, b_T
      if not r in *bindings* **then**
        store $r \rightarrow r_T$ in *bindings*
      else if *operation* = store and $r_T \in$ vectors and $a_T < r_T$ **then**
        try repairWeakStore
      else if *operation* = load and $r_T \in$ maps and $b_T < in(r_T)$ **then**
        try repairWeakMap
      else if $r_T \neq$ lookup r in *bindings* **then**
        try rewriting wildcard masks with declared masks to unify masksets
        if rewrite not possible **then**
          Abort with an error
        end if
      end if
      if $r_T <$ max a_T, b_T **then**
        try repairAncs
      end if
    end for
  end while

```
Sbox, xtime : Byte -> Byte
key : secret<a> vector of Byte(4)
def mixcols( in:public vector of Byte(4) ) : secret<X> vector of Byte(4)
{
out : secret<X> vector of Byte(4) )
temp : vector of Byte(4)
 for i in range(4)
   temp[i] := Sbox[ in[i]^key[i] ]
 for i in range(4)
   out[i] := xtime[temp[i]] + temp[(i+1)%4] + xtime[temp[(i+1)%4]] +
             temp[(i+2)%4] + temp[(i+3)%4]
 return out
}
```

The type `Byte` is used to represent concrete data, while the higher-order type `vector` is used to indicate logical grouping. Unlike C the use of a aggregate type does not imply anything about the representation in memory, and is simply a convenience for the programmer [11]. Each type is annotated by a security level. If the variable is already known to the attacker and can be freely revealed the annotation is `public`. When the variable must remain hidden a set of masks is specified with the `secret` annotation. In the example each declared set is a singleton although larger sets are inferred for temporary variables during compilation. The programmer has specified the existence of two masks in the source-code:

1. The key is covered by a mask `a`, as this masking operation must have occurred prior to the execution of the mixcols procedure this named mask forms part of an interface with the calling code.
2. The return value is covered by a mask `X`, again this forms part of an interface with the calling code.

Any variable without a security annotation is initially assumed to be public. If this assumption causes an error during the inference stage then it will be rewritten with a more secure annotation.

Both the `Sbox` and `xtime` functions are declared as public mappings from `Byte` to `Byte`. This leaves some flexibility in their definition, previous work [11] shows how declarative definitions can be provided and memorized into lookup-tables by a compiler, or the program can supply a constant array of bytes to encode the mapping.

The algorithm operates according to the process in Algorithm 1. We now illustrate some of the steps involved in iterating the inference and repair processes. Although the algorithm operates on the low-level 3-op form of the code this description will proceed at a source-level for reasons of space.

5.1 WeakMap detected in `Sbox`

Type inference fills in intermediate types until it encounters the expression `Sbox[ in[i]^key[i] ]`. The type of `in[i]^key[i]` is inferred to be `secret<a>`, while the declaration of `Sbox` is of type `Byte -> Byte`. The compiler can "repair" this type error by synthesizing a new copy of `Sbox`, which we will call `Sboxm`. As this expression includes a part with a secret tag the minimum type annotation for `Sboxm` is given by `secret<x> Byte -> secret<y> Byte` for some secure `x` and `y` such that `x` $\neq$ `y`. As the index expression is masked under `a` we can insert a new mask to produce an annotation of `secret<a> -> secret<b>` for some fresh mask `b`. This

mask is called a wildcard as we may merge it with other masks later to reduce the number of random values required.

The new table `Sboxm` must be generated at runtime from the original `Sbox` table and the masks. The compiler inserts the following code:

```
Sboxm : secret<a> Byte -> secret<b> Byte
b : fresh Byte;
for i in range(256) :
 Sboxm[i] := Sbox[i^a] ^ b
```

5.2 WeakStore detected in `temp`

As the programmer did not specify a security annotation for `temp` it defaulted to public. The output of the `Sbox` map is annotated by `secret<b>`. This causes an error in the inference as a secret value cannot be stored in a public variable. The compiler fixes this error by altering the declared type of `temp` to be `secret<b>`. This step is valid as it is always sound to increase the security of a variable. The compiler now expands temporaries in the expression evaluation and converts the access to use the masked table:

```
temp : secret<b> vector of Byte(4)
t  : secret<a> Byte
t2 : secret<b> Byte
for i in range(4) :
 t      := in[i] ^ key[i]
 t2     := Sboxm[t]
 temp[i] := t2
```

5.3 Revelation detected in second loop

The algorithm proceeds into the second loop where it tries the following inference until it reaches an error:

```
t3 : secret<c> Byte     // Compiler inserted
t4 : Byte               // Compiler inserted
for i in range(4)
 t3 := xtime[temp[i]]
 t4 := t3 ^ temp[(i+1)%4]
```

The error arises because the type of `t3` is declared to be the same type inferred for the expression `temp[(i+1)%4]`. The inference rules for an xor operation cancel out masks that appear on both sides producing the public annotation for t4. As a variable predecessor in the UDC is annotated secret this constitutes a revelation error. The compiler uses the process described in Section 4.2 to decide upon a repair. As one predecessor `temp` is a vector it would be more costly to flip the masks uniformly in each element, rather than simply flip the masks on `t3`. As flipping the existing masks does not produce a solution the compiler inserts a new wildcard mask `b`.

```
t3,t5 : secret<c,b> Byte
t4,t6 : secret<b>   Byte
out : secret<c,b> Vector of Bytes(4)
```

```
for i in range(4)
 t3 := xtime[temp[i]] ^ b            // Inserted flip operation
 t4 := t3 ^ temp[(i+1)%4]
 t5 := t4 ^ xtime[temp[(i+1)%4]]
 t6 := t5 ^ temp[(i+2)%4]
 out[i] := t6 ^ temp[(i+3)%4]
```

5.4 Mask collisions detected

Two subsequent iterations detect inequalities in the masking sets. These are resolved by unifying the wildcard mask `c` with the declared output mask `X` and inserting a flip operation to remove the mask `b` from the final result.

```
t3,t5 : secret<X,b> Byte
t4,t6 : secret<b>   Byte
t7 : secret<X,b>
out : secret<X> Vector of Bytes(4)
for i in range(4)
 t3 := xtime[temp[i]] ^ b
 t4 := t3 ^ temp[(i+1)%4]
 t5 := t4 ^ xtime[temp[(i+1)%4]]
 t6 := t5 ^ temp[(i+2)%4]
 t7 := t6 ^ temp[(i+3)%4]
 out[i] := t7 ^ b;
```

6 Application in practice

Our discussion above demonstrates the working principle of our algorithm. We explain how one would transform an insecure (i.e. unmasked) description of AES (reduced to `SubBytes` and `MixColumns` for the sake of brevity) to secure code (i.e. masked). In this section we briefly explain how this is translated to ARM assembly language [12] and compare the performance of the automatically-masked code with masked code that has been hand-coded. We then show the results of some DPA experiments conducted with this code.

6.1 ARM assembly version and comparison

A prototype compiler was implemented in Haskell that reads the program source and outputs ARM assembly compatible with the Crossworks tools [13]. The use of a declarative language makes the implementation of a rule-based type-checker particularly simple, Haskell in particular is suited to embedding experimental languages due to the presence of monad transformers and their ability to add new forms of control flow.

An extract of this conversion, including annotation of masks as comments denoted `a` and `b`, is given below:

```
PUSH  {R3-R12,R14}
LDR   R5, =in
LDR   R6, =key
LDR   R7, =Sbox_M
LDR   R8, =temp
```

```
LDR   R9, =xtime_M
LDR   R10, =out

LDRB  R0, [R5, #0]    // [[]], [[]], ?
LDRB  R2, [R6, #0]    // [[a]], [[a]], ?
EOR   R3, R0, R2      // [[a]], [[]], [[a]]
LDRB  R0, [R7, R3]    // [[b]], [[a],[b]], [[a]]
STRB  R0, [R8, #0]    // [[b]], ?, [[b]]
```

This code snippet shows the sequence of assembly instructions from pushing some registers onto the stack when the function is called, to loading the first byte of the AES state and key, exclusive-oring these two bytes and using them as index for the `SubBytes` operation. As described before, a masked `SubBytes` table must be generated each time the AES code is executed (to facilitate readability this is however not included in the code shown here). Then the result of the `SubBytes` operation is stored. The code is annotated with comments that show how each register is masked for each instruction, where the mask is given provided between the brackets `[[ ]]`. As we assume that the plaintext is public (i.e. unmasked) and the key is secret (i.e. masked), the first line which refers to loading the plaintext shows an empty masking set. The second line which refers to the loading of the key shows that the mask `a` is used. In the third line where input and key are exclusive-ored, the result inherits the mask from the key. The fourth line, which refers to the `SubBytes` operation, show that this operation has a masked input (mask `a` is used) and maps this input to a value which is masked differently (mask `b` is used).

For comparison purposes we provide two more code snippets, the left-hand one showing an implementation which was hand-coded and uses loops, the right-hand one showing an implementation which was hand-coded and unrolls these loops:

```
   PUSH  {R3-R12,R14}
   BL    SubBytes
// ----------------------------
// SubBytes
// Input : R1 - pointer to data
// Output : @R1
// ----------------------------
SubBytes:
   MOV   R5, #4
   LDR   R6, =acAESsbox
SubBytes:
   SUB   R5, R5, #1
   LDRB  R7, [R1, R5]
   LDRB  R8, [R6, R7]
   STRB  R8, [R1, R5]
   CMP   R5, #0
   BNE   SubBytes
   BX    LR
```

```
// ------------------
// Macros
// ------------------

.macro  Msub   i=0
   LDRB  R5, [R0, \i]
   LDRB  R5, [R4, R5]
   STRB  R5, [R0, \i]
.endm




   PUSH  {R3-R12,R14}
   LDR   R4, =acAESsbox
Ssub_s0:  Msub   #0
```

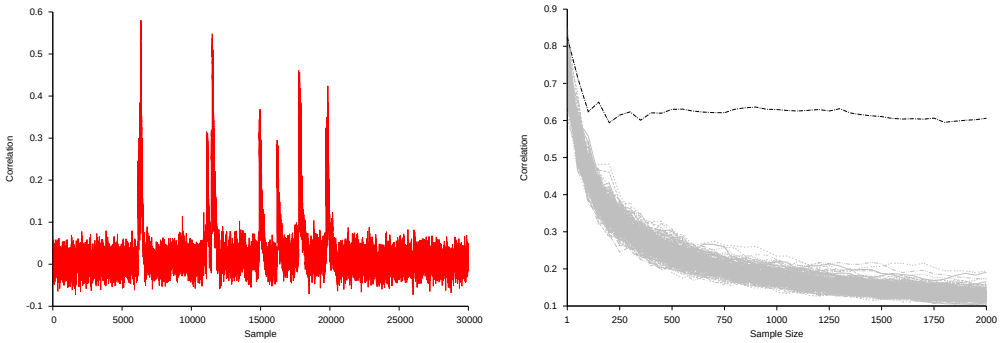
There is a clear difference in coding styles which of course leads to different performance figures. The hand-coded assembly version with loops requires 147 clock cycles to copmute the `SubBytes` and `MixColumns` function (for one column of the state only), whereas an unrolled version (hand-coded) requires 52 clock cycles. Our algorithm that automatically adds the masking to an unmasked implementation produces code that requires 76 clock cycles.

6.2 Results of a DPA

Naturally, one needs to know whether the security against first-order attacks that we argue is provided by our algorithm also translates into practice. Otherwise it would be imprudent to use any code in a secure device. Whilst we have argued that our algorithm checks the necessary conditions for a masked implementation to be secure it is left to show that for a 'simple' device (i.e. a device with Hamming weight leakage), it is left as an open problem to show that this translates into a countermeasure that prevents DPA on devices where the leakage corresponds to another model.

In the remainder of this section we describe some experiments that we conducted on an ARM7TDMI microprocessor [12] using the example described in Section 5. We use a simple experimental board on which such a microprocessor is mounted to acquire power traces using a 'standard' setup: a differential probe is used to acquire power traces. We use a suitable sampling frequency and have an artificially generated trigger point which eases the alignment of traces.

The first experiments focused on compiling the unmasked code described in Section 5 using a standard C compiler. In our case this was Crossworks for ARM. We acquired 2000 traces showing the power consumption during the execution of the code, for a constant secret key and a randomly generated input.



(a) A correlation trace using 2000 traces.

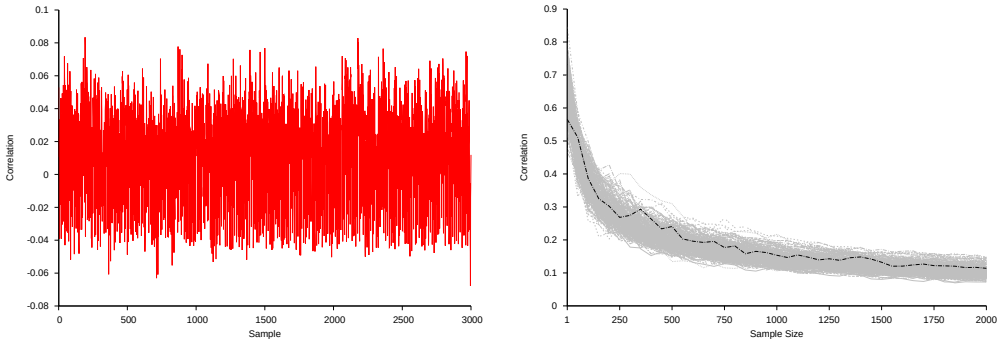
(b) The maximum correlation plotted against the number of traces. The correct hypothesis plotted in black and the incorrect hypotheses in gray.

Fig. 1. Results of a DPA attack on the unmasked implementation which give clear evidence for the vulnerability of such an implementation on the target platform.

The correlation was computed between the output of one of the substitution tables for all possible key hypotheses. The correlation trace for the correct hypothesis is shown in the left panel of Fig. 1, where numerous peaks show at what points in time the result of the substitution table is processed by the microprocessor. The right panel shows that as little as 100 acquisitions would be necessary to reveal the correct key byte.

We performed the same analysis on the code masked by our algorithm and translated into ARM assembly. Fig. 2 shows the results. There some peaks in the correlation trace (left panel) but they are caused by noise in the acquisitions and unrelated to the manipulation of values being manipulated by the microprocessors. In addition, the right panel shows that peaks

in traces of incorrect key hypotheses are equally significant which means that no distinction between key hypotheses is possible. This confirms that the executed code is resistant to DPA attacks on this platform.



(a) A correlation trace using 2000 traces.

(b) The maximum correlation plotted against the number of traces. The correct hypothesis plotted in black and the incorrect hypotheses in gray.

Fig. 2. Results of a DPA attack on the masked implementation demonstrating that no information leaks.

7 Conclusion

In this paper we detail an algorithm for the automated generation of code that is resistant to first-order DPA, and provide a detailed example of how this algorithm can be implemented. While the source code needs to be written in a particular format, a developer does not need to have a detailed knowledge of the assembly language of a given microprocessor. Indeed, given that our compiler produces code that is comparable to assembly code written by a human, one could use the same source for numerous platforms reducing development cost considerably.

The current version assumes that the target microprocessor leaks information independently for each instruction executed. Some devices may leak information in a different model, where the information leakage depends on consecutive instructions. This may impose a further restriction on the compiler, i.e. that variables masked with the same mask cannot be manipulated in adjacent instructions, and subsequent iterations of our compiler will seek to address this issue.

Our compiler was designed to produce code that would be resistant to first-order DPA. One would therefore expect the code produce to be vulnerable to higher-order DPA. Our compiler seeks to reduce the number of masks that are used, which is what would allow higher-order DPA to be performed. If one took a more counterintuitive approach and did not seek to optimize the number of masks that are used, the result would be a mask that evolved throughout the computation of an algorithm. This would provide some resistance to higher-order DPA. However, this issue is beyond the scope of this paper will be evaluated in a subsequent iteration of our compiler.

References

1. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium. (1998) 63–78
2. Agat, J.: Type based techniques for covert channel elimination and register allocation. PhD thesis, Chalmers University of Technology (2001)
3. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Won, D.H., Kim, S., eds.: ICISC 2005. Volume 3935 of LNCS., Springer (2006) 156–168
4. Lux, A., Starostin, A.: A tool for static detection of timing channels in Java. In: Constructive Side-Channel Analysis and Secure Design (COSADE), CASED (2011) 126–140
5. Regazzoni, F., Cevrero, A., Standaert, F.X., Badel, S., Kluter, T., Brisk, P., Leblebici, Y., Ienne, P.: A design flow and evaluation framework for DPA-resistant instruction set extensions. In Clavier, C., Gaj, K., eds.: CHES 2009. Volume 5747 of LNCS., Springer (2009) 205–219
6. Tiri, K., Verbauwhede, I.: A digital design flow for secure integrated circuits. IEEE Trans. on CAD of Integrated Circuits and Systems **25**(7) (July 2006) 1197–1208
7. Zdancewic, S.A.: Programming Languages for Information Security. PhD thesis, Cornell University (August 2002)
8. Blömer, J., Guajardo, J., Krummel, V.: Provably secure masking of AES. In Haddad, H., Omicini, A., Wainwright, R.L., Liebrock, L.M., eds.: SAC 2004, ACM (2004) 69–83
9. Oswald, E., Mangard, S., Pramstaller, N., Rijmen, V.: A side-channel analysis resistant description of the AES S-box. In Gilbert, H., Handschuh, H., eds.: FSE 2005. Volume 3557 of LNCS., Springer (2005) 413–423
10. Barbosa, M., Moss, A., Page, D., Rodrigues, N., Silva, P.F.: A domain-specific type system for cryptographic components. In: Fundamentals of Software Engineering (FSEN). (2011)
11. Moss, A., Page, D.: Bridging the gap between symbolic and efficient AES implementations. In Gallagher, J.P., Voigtländer, J., eds.: Partial Evaluation and Program Manipulation (PEPM), ACM (2010) 101–110
12. ARM7TDMI technical reference manual. Available at: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html`
13. Crossworks for ARM. `www.rowley.co.uk/arm/`