HAL open science

Can code polymorphism limit information leakage?

Antoine Amarilli, Sascha Müller, David Naccache, Daniel Page, Pablo Rauzy, Michael Tunstall

To cite this version:

Antoine Amarilli, Sascha Müller, David Naccache, Daniel Page, Pablo Rauzy, et al.. Can code polymorphism limit information leakage?. 5th Workshop on Information Security Theory and Practices (WISTP), Jun 2011, Heraklion, Crete, Greece. pp.1-21, 10.1007/978-3-642-21040-2_1 . hal-01110259

HAL Id: hal-01110259

https://inria.hal.science/hal-01110259v1

Submitted on 27 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Can Code Polymorphism Limit Information Leakage?

Antoine Amarilli[1], Sascha Müller[2], David Naccache[1], Daniel Page[3], Pablo Rauzy[1], and Michael Tunstall[3]

[1] École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris Cedex 05, France.
{name.surname}@ens.fr
[2] Technische Universität Darmstadt, Security Engineering
Hochschulstraße 10, D-64289 Darmstadt, Germany.
mueller@seceng.informatik.tu-darmstadt.de
[3] University of Bristol
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK.
{page,tunstall}@cs.bris.ac.uk

Abstract. In addition to its usual complexity assumptions, cryptography silently assumes that information can be physically protected in a single location. As one can easily imagine, real-life devices are not ideal and information may leak through different physical side-channels. It is a known fact that information leakage is a function of both the executed code F and its input x.

In this work we explore the use of polymorphic code as a way of resisting side channel attacks. We present experimental results with procedural and functional languages. In each case we rewrite the protected code code F_i before its execution. The outcome is a genealogy of programs $F_0, F_1, \ldots$ such that for all inputs x and for all indexes $i \neq j \Rightarrow F_i(x) = F_j(x)$ and $F_i \neq F_j$. This is shown to increase resistance to side channel attacks.

1 Introduction

From a security perspective, the advent of a software monoculture is an oft cited problem. Monoculture software is coarsely defined as programs (*e.g.*, Internet Explorer), generated from the same source code by the same compiler (*e.g.*, Visual Studio) and executed on the same processor family (*e.g.*, Intel x86) under control of the same operating system (*e.g.*, Windows). The premise is that monoculture makes attacks easier: an attack against any one member can be applied directly to the entire population; analogues exist in biology where monocultures and parthenogenesis are known to ease the spread of disease and lessen adaptation to environmental changes.

Various (seemingly different) protections attempt to break software monoculture through diversification. A simple example is that of Address Space Layout

Randomization (ASLR): if each program is executed within a different address offset, then any assumptions an opponent makes on the location of a particular datum limits attacks to a smaller subset of the population.

This argument is equally relevant to embedded security and side-channel resilience, even if one accepts Kerckhoffs' principle (that cryptographic security should lay in the key alone). Defenders endeavor to make attacks harder by randomizing execution: even if opponents know the program being executed, their task of exploiting leakage is harder since they cannot focus their analysis with accuracy.

Background and Related Work Focusing specifically on temporal randomization (*e.g.*, ignoring masking and related techniques), the term desynchronization is often used. The basic idea is to somehow alter normal program execution by "shuffling" instructions on-the-fly. This roughly means that the i-th instruction is no longer executed during the i-th execution cycle: the defender either swaps it with another instruction or inserts delays that cause the code to execute during a j-th cycle.

Consider the following randomization of the AES S-box layer [5]. Assuming that `SBOX` is a pre-computed table representing the AES S-box, a simple C implementation might resemble the following loop:

```
for( int i = 0; i < 16; i++ ) {
   S[ i ] = SBOX[ S[ i ] ];
}
```

To randomise the order of accesses to `SBOX`, one idea would be to maintain a table `T` where the i-th entry, *i.e.*, `T[i]`, is initially set to `i` for $0 \leq \texttt{i} < 16$. This table can be used for indirection as follows:

```
for( int i = 0; i < 16; i++ ) {
   S[ T[ i ] ] = SBOX[ S[ T[ i ] ] ];
}
```

Note that this represents what one might term *online* overhead in the sense that the indirection's cost is paid during every program execution. Of course the trade-off is that table `T` can be updated, more specifically randomized, at regular intervals (*e.g.*, after each execution of the program) to ensure that S-box accesses are reordered. Such *re-randomization* is achievable using something as simple as:

```
t = rand() & 0xF;

for( int i = 0; i < 16; i++ ) {
   T[ i ] = T[ i ] ^ t;
}
```

This update of `T` represents what we term *offline* overhead: although the computational toll is not paid before run-time, the cost is offline in the sense that it is not borne during the execution of the program itself. Related work includes (but is certainly not limited to):

- Herbst *et al.* [7] describe the use of "randomization zones" within an AES implementation; the basic idea is to randomly reorder instructions within selected round functions, thus temporally skewing them in an execution profile.
- May *et al.* [12] describe NONDET, a processor design idiom that harnesses Instruction Level Parallelism (ILP) within a given instruction stream to issue instructions for execution in a random (yet valid) order. This essentially yields a hardware-supported and hence more fine-grained and more generic, version of the above.
- A conceptually similar idea is the design of re-randomizable Yao circuits by Gentry *et al.* [6]; the goal in both cases is to prevent leakage and in a sense construct per-use programs (circuits) via randomization.
- Many proposals have made use of random timing delays, *i.e.*, temporal skewing. For example Clavier *et al.* [3] describe a method which uses the interrupt mechanism while Tunstall and Benoît [16] describe a similar software-based approach.
- A vast range of program obfuscation techniques have appeared in the literature (see [4] for an overview) and are used in industry. The typical goals are to make reverse engineering harder and diversify the form of installed software; other applications include the area of watermarking.

Goals and Contribution We consider the use of program self-modification as a means to allow a more general-purpose analogue of the above; we aim to describe an approach which

1. can be automated in a compiler (noting that parallelizing compilers can already identify light-weight threads in programs), and
2. can be composed with other countermeasures (*e.g.*, masking).

The basic idea is to (randomly) rewrite the program before execution and limit all overhead to being offline. The online overhead would be essentially nil: the overhead is associated purely with the number of static instructions in the program, *i.e.*, the cost of rewriting, rather than the number of dynamic instructions

executed. A fully randomized rewriting approach would be costly since it demands analysis and management of instruction dependencies: in a sense, this would be trying to do in software what a NONDET processor does in hardware. We nonetheless explore this approach concretely using Lisp in Section 5. A conceptually simpler approach would be to follow the reasoning of Leadbitter [11] who imagines randomization as choices between threads which are *already* free from dependencies.

More formally, letting $c = F_0(k, m)$ denote a cryptographic program run on public input m and secret input k in a protected device, we explore ways to code F_0 in such a way that after each execution F_i will rewrite itself as F_{i+1} whose instructions differ from those of F_i before returning c.

In other words $\forall i, j, m, k, \quad F_i(k, m) = F_j(k, m)$, but $i \neq j \Rightarrow F_i \neq F_j$, a property that makes the attacker's signal collection task much harder.

2 Algorithmic Description

We represent the straight-line program fragment under consideration as a graph with n *levels*; each level contains a number of nodes which we term *buckets*. G_i denotes a list of buckets at the i-th level, with $|G_i|$ giving the length of this list. $G_{i,j}$ denotes a list of instructions in the j-th bucket at the i-th level, with $|G_{i,j}|$ giving the length of the list and $G_{i,j}[k]$ giving the k-th instruction.

Consider two instructions $\texttt{ins}_1$ and $\texttt{ins}_2$ that reside in buckets at the i-th and j-th level respectively: $\texttt{ins}_1$ *may* be dependant on $\texttt{ins}_2$ iff $i > j$, $\texttt{ins}_1$ and $\texttt{ins}_2$ *must* be independent if $i = j$. Informally, levels represent synchronization points: for some $i > j$, no instruction within the i-th level can be executed until every instruction within the j-th level has been executed. As such, buckets within a level can be viewed as threads and each level as a thread team: for some $i > j$ and k, instructions within buckets i and j can execute in parallel (or constituent instructions be scheduled in any order) if both buckets are at level k.

Our approach is to maintain in memory two copies of program instructions: a static version (*source program*) and a dynamic version (*target program*). The target program is actually executed at run-time. At some parameterized interval, the target program is rewritten using instructions extracted from the source program. The rewriting process is driven by the program graph which describes the source program structure: the goal is to use the structure to randomize the order according to which instructions are written into the target program while preserving dependencies. This is possible since the layer and buckets are essentially a pre-computed description of instructions inter(in)dependencies. The rewriting process is performed at run-time with a granularity matching the level of execution randomization (which relates loosely to the level of security) dictated by the context.

Algorithm 1: initializes the indirection lists driving the program rewriting process.

Input: A program graph G with n levels representing a source program S.

```
for i = 0 upto n − 1 do
    Let t be the type of buckets within G_i.
    R_i ← ∅
    if t = 1 then
        for j = 0 upto |G_i| − 1 do
            Append j to the list R_i.
        end for
    end if
    else if t = 2 then
        for j = 0 upto |G_i| − 1 do
            for k = 0 upto |G_{i,j}| − 1 do
                Append j to the list R_i.
            end for
        end for
    end if
end for
```

2.1 Bucket Types

To facilitate rewriting we define two bucket-types. Where we previously denoted a bucket as $G_{i,j}$ we now write $G^t_{i,j}$ for a type-t bucket. Consider two buckets G and G', both at level i in the program graph:

Type-1 if the bucket is of type-1 this means we must extract *all* instructions in one go. This ensures that if we select G and then G', the instructions from G are written in a contiguous block within the target program and *then* instructions from G' are written in a second contiguous block.

Type-2 if the bucket is of type-2 this means we can extract a *single* instruction at a time. This means that if we select G and then G', instructions can be freely interleaved with each other.

The two bucket types represent a tradeoff. On one hand, using type-2 buckets is ideal since it allows fine-grained interleaving of instructions and therefore a higher degree of randomization. However, to preserve the program's functional behavior, such buckets must use a disjoint set of registers so that the instructions can be freely interleaved. Since a given register file is limited in size, this is sometimes impossible; to avoid the problem, one can utilize type-1 buckets as an alternative. Here, register pressure is reduced since buckets can use an overlapping set of registers.

2.2 Rewriting Algorithms

One can remove the restriction at extra cost, but to simplify discussion assume that all buckets at a particular level in the program graph are of the same type.

Algorithm 2: randomly rewrites the source program into a target program.

Input: A program graph G with n levels representing a source program S.
Output: The target program T representing a valid, randomized reordering of instructions from S.

```
Set T ← ∅
for i = 0 upto n − 1 do
    Shuffle the list R_i.
end for
for i = 0 upto n − 1 do
    Let t be the type of buckets within G_i.
    if t = 1 then
        for j = 0 upto |R_i| − 1 do
            j' ← R_i[j]
            for k = 0 upto |G_{i,j'}| − 1 do
                Let I be the next unprocessed instruction in G_{i,j'}.
                Append I to the target program T.
            end for
        end for
    end if
    else if t = 2 then
        for j = 0 upto |R_i| − 1 do
            j' ← R_i[j]
            Let I be the next unprocessed instruction in G_{i,j'}.
            Append I to the target program T.
        end for
    end if
end for
return T
```

To drive the rewriting process, Algorithm 1 is first used to initialize n *indirection lists*: R_i is the i-th such list whose j-th element is denoted $R_i[j]$. This effectively sets $R_i = \langle 0, 1, \ldots, |G_i| - 1 \rangle$ if the buckets within G_i are of type-1, or

$$R_i = \langle \underbrace{0, 0, \ldots, 0}_{|G_{i,0}| \text{ elements}} , \underbrace{1, 1, \ldots, 1}_{|G_{i,1}| \text{ elements}} , \ldots \rangle$$

if buckets are of type-2. The lists relate directly to table `T` used within the example in Section 1.

When the program needs to be rewritten, Algorithm 2 is invoked: one level at a time, instructions from the source program S are selected at random, driven by the associated indirection list, to form the target program T. Note that before this process starts, each indirection list is randomly shuffled; this can be done, for example, by applying a Fisher-Yates shuffling [9, Page 145-146] driven by a suitable LCG-based PRNG [9, Page 10-26].

3 Concrete Implementation

As an example, consider an AES implementation [5] using an eight-bit software data-path; we represent the state matrix using a sixteen-element array `S`. One can describe instructions that comprise a standard, non-final AES round (*i.e.*, `SubBytes`$\frown$`ShiftRows`$\frown$`MixColumns`$\frown$`AddRoundKey`), as a program graph. Using both C and continuation dots for brevity, such a program graph is shown in Figure 1.

- For the example (and in general) one can specialize the program rewriting algorithm once the source program is fixed. *e.g.* all loops can be unrolled, empty levels or levels with a single bucket can be processed at reduced cost and special case treatment can be applied to levels with a single bucket type.
- The `MixColumns` layer houses buckets that can be split into smaller parts depending on register pressure. *e.g.* within level five one could split the second phase of each bucket into a further layer with sixteen buckets: each bucket would compute one element of the resulting state matrix (rather than the current formulation where four buckets each compute four elements).

4 Experimental Evaluation

Algorithm 2 was implemented on an ARM7 microprocessor. In this section we describe the performance of an unrolled reordered AES implementation and how one could attack such an implementation. We compare this with a straightforward unrolled AES implementation.

4.1 Performance

A standard (unprotected) unrolled AES implementation and a polymorphic AES code were written for an ARM7 microprocessor. The polymorphic version is only 1.43 times slower than the unrolled AES (7574 cycles *vs.* 5285), a time penalty which is not very significant for most practical purposes. This comparison is only indicative as faster polymorphic programs are possible (our rewriting function was written in C with no optimizations). Nonetheless, the polymorphic AES code requires a significant amount of extra RAM which might be problematic on some resource constrained devices.

4.2 Attacking a Standard AES Implementation

A standard AES code will call each sub-function deterministically. This typically involves constructing a loop that will go through all the indexes required to compute a given function in a fixed order. These loops are typically seen in the

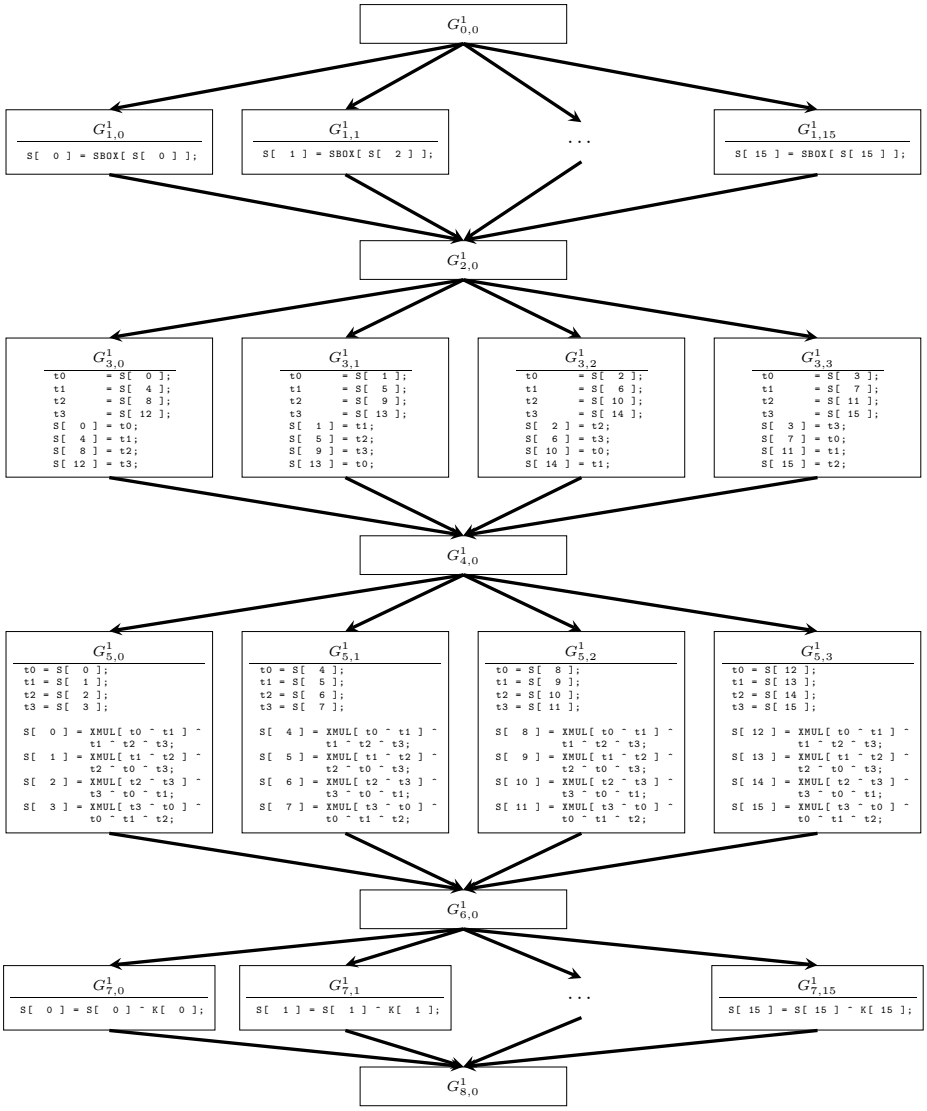

Fig. 1. A program graph for one AES round; the graph consists of 9 levels, with levels 0, 2, 4, 6 and 8 acting as synchronization points. Note that `SBOX` and `XMUL` represent precomputed tables for the AES S-box and `xtime` operation.

instantaneous power consumption, as a pattern of nine distinct patterns corresponding to the AES' first nine rounds. The last round is typically represented by a different pattern because of the absence of the `MixColumn` function.

The different sub-functions of an AES code can be identified by inspecting a power consumption trace. In the left hand part of Figure 2 two patterns of sixteen peaks can be seen. These correspond to the plaintext and secret key being permuted to enable efficient computation given the matrix representation in the AES' specification. This is followed by a pattern of four peaks that correspond to the exclusive or with the first key byte (the ARM7 has a 32-bit architecture). Following this there is a pattern of sixteen peaks that corresponds to the `SubBytes` function and two patterns of four peaks that correspond to the generation of the next subkey. `ShiftRow` occurs between these functions but is not visible in the power consumption. The exclusive or with this subkey can be seen on the right hand side of Figure 2, which means that the remaining area between this exclusive or and the generation of the subkey is where `MixColumn` is computed. However, no obvious pattern can be seen without plotting this portion of the trace with a higher resolution.

It is known that power consumption is typically proportional to the Hamming weight of the values being manipulated at a given point in time. This can be used to validate hypotheses on portions of the secret key being used in a given instance [2,10]. For example, the correlation between the Hamming weight of `SubBytes`'s output the power consumption traces can be computed in a pointwise manner for a given key hypothesis, in this case we only need to form a hypothesis on one secret key byte. If the hypothesis is correct a significant correlation will be visible as shown in the right hand graphic of Figure 2, we note that the maximum correlation coefficient is ~ 0.6. If the key hypothesis is incorrect then no significant hypothesis will be present.

1000 encryption power consumption traces were taken where the secret key was a fixed value and the plaintext randomly changed for each trace. The right hand graphic of Figure 2 shows a trace of the correlation between the points of the power consumption traces and the Hamming weight of the result of the first byte produced by the `SubBytes` function given that the secret key is known. That is, the correlation is computed between the list of Hamming weights and the values of the first point of each trace, the values of the second point of each trace, *etc.* to form a trace of correlation values. The first peak corresponds to the point in time at which the first byte is produced in `SubBytes` and indicates which of the sixteen peaks corresponds to that byte being produced. The subsequent peaks in the correlation trace indicate the instants where the same byte is manipulated by `MixColumns`.

4.3 Attacking an Unrolled AES Implementation

The typical power consumption trace of an unrolled AES is shown in the left part of Figure 3.

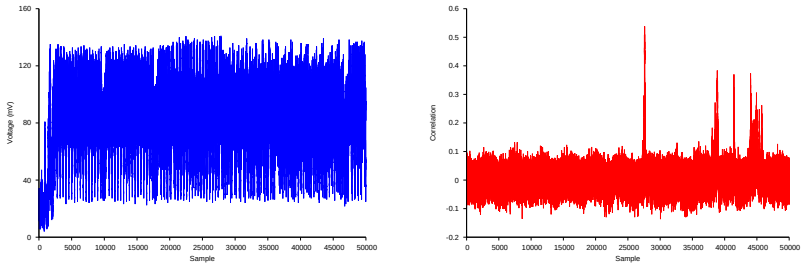

Fig. 2. Power consumption trace of a single round of an AES encryption performed by an ARM7 microprocessor (left) and a Differential Power Analysis signal (right).

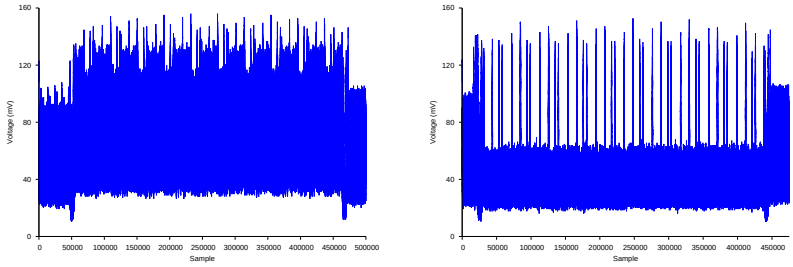

Fig. 3. Power consumption trace of an unrolled AES on ARM7. Unprotected (left) and polymorphic (right) codes.

In Figure 4, we note that the maximum correlation coefficient for an unrolled implementation is ~ 0.7.

Figure 4 is the analogous of 2 under identical experimental conditions (1000 traces *etc*). Interpretation remains the same: the subsequent peaks in the correlation trace indicate the instants at which the same byte is manipulated in `MixColumns` but have a lower correlation coefficient.

The left graph of Figure 6 shows the maximum observed correlation for all 256 possible hypotheses for one observed key for x power consumption traces. Incorrect hypotheses are plotted in grey and the correct hypothesis is plotted in black. We note that ~ 100 traces suffice to distinguish the correct hypothesis from the incorrect hypotheses.

4.4 Attacking a Polymorphic AES Implementation

A polymorphic AES, as described in Section 3 was implemented on an ARM7. The power consumption for the round function changes considerably. In the implementation, the subset of opcodes used has a lower average power consumption and features local peaks in the power consumption caused by the call and return from the subfunctions in the individual round (right part of Figure 3). The individual round functions can only be identified by the time required to compute them as the patterns visible in Figure 2 are no longer present. These peaks can easily be removed by implementing the round function as one function. However, this feature is convenient for our analysis.

Figure 5 is the equivalent of Figure 4 for a polymorphic AES. Figure 5 features two groups of peaks, the first of which has a correlation of ~ 0.06; this group is caused by the sixteen possible positions where the byte output from the `SBOX` indexed by the exclusive-or of the first plaintext byte and the first byte of the secret key is created. A second series of peaks representing a correlation of ~ 0.1 is visible. This series of peaks is caused by the sixteen possible positions where the same byte can be manipulated in the `MixColumn` function. We can note that these correlation coefficients are very low and $20,000$ power consumption curves were required to produce a correlation coefficient that is significantly larger than the surrounding noise.

The right graph of Figure 6 shows the maximum observed correlation for all 256 possible hypotheses for one observed key for x power consumption traces. The incorrect hypotheses are plotted in grey and the correct hypothesis is plotted in black. We note that $\sim 2,500$ traces are required to distinguish the correct hypothesis from the incorrect ones. This is considerably more than required to distinguish the correct hypotheses when attacking a non-polymorphic AES.

5 Can Lisp-Like Languages Help?

A further sophistication step consists in requiring F_i and F_{i+1} to have an extreme difference. While we do not provide a rigorous definition of the word *extreme*,

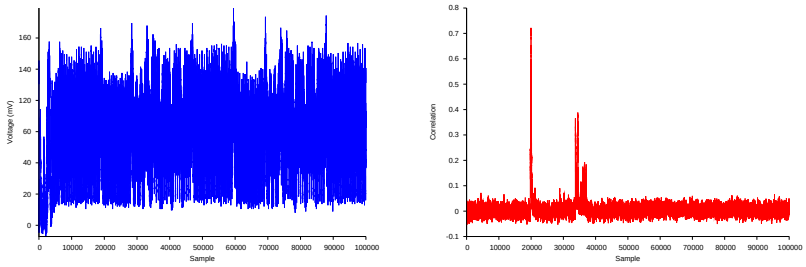

Fig. 4. The rightmost trace is the correlation between the power consumption and the output of the S-box that operates on the $\oplus$ of the first plaintext byte and the secret key. The leftmost trace shows a sample power consumption, in millivolts, during the same period of time.

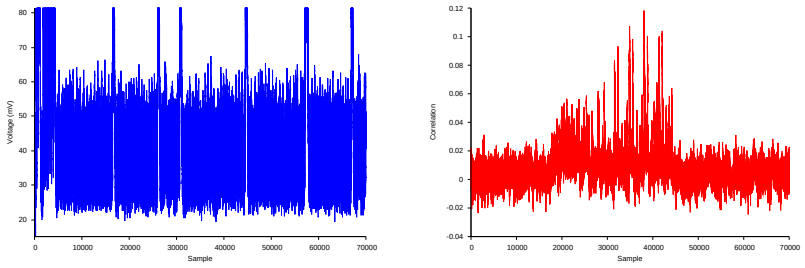

Fig. 5. The analogous of Figure 4 for the polymorphic code. In the leftmost trace the round functions are divided up by peaks in the power consumption.

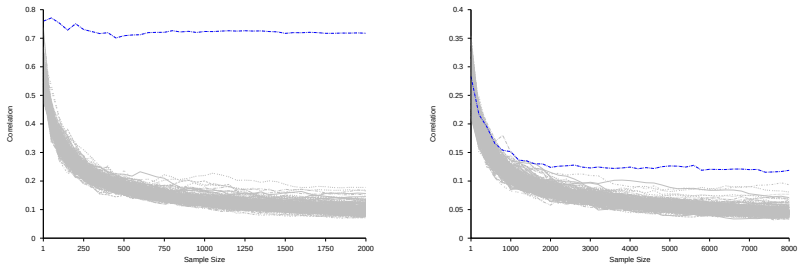

Fig. 6. Maximum correlation. Unrolled AES (left curve) and unrolled polymorphic AES (right curve).

the aim of our next experiment, nicknamed PASTIS, is to illustrate the creation of a program able to rewrite itself in a way that does not alter functionality but potentially changes *all its code*. We call a code *fully polymorphic* if any instruction of F_i can potentially change in F_{i+1}.

The code was designed with two goals in mind: illustrate the way in which fully polymorphic code is designed and provide a platform allowing to comfortably test the efficiency of diverse replacement rules as a step stone towards the design of a fully polymorphic AES code.

Such techniques can already be seen in polymorphic viruses as a way to foil signature-based detection attempts by anti-virus software; they also appear in code obfuscation systems. Readers can refer to [17] for more information on this topic.

PASTIS is written in Scheme for the MIT Scheme implementation [15]. The payload to transform (*e.g.* an AES) also has to be written in Scheme and is restricted to the subset of the Scheme syntax which the rewriting system is able to understand (Of course, since the rewriting engine has to rewrite itself, it is itself written using this limited subset of Scheme).

PASTIS is modular in a way making it easy to install new rewriting rules. Rules must change the source code's form while keeping it functionally equivalent. In this paper we voluntarily provide illustrative rules which could not work indefinitely because they tend to make the size of the code increase.

5.1 Structure

The top-level PASTIS function is `pastis-generator`. It creates the self-rewriting program from the payload and a rewriting function (which takes code as input and produces functionally equivalent rewritten code as its return value).

The produced code behaves functionally like the payload function: it will be evaluated to the same value if it gets the same parameters. However, it will additionally print, during the evaluation, a rewritten, equivalent version of itself.

Of course, the rewritten version is still functionally equivalent to the original payload and will also produce a rewritten version of itself, which, in turn, can be run, and so on, *ad infinitum* (forgetting about the growing size of the rewritten code, *i.e.*, assuming that we have an infinite amount of memory).

Internal Structure The use of the `pastis-generator` function is quite straightforward; its role is to provide a convenient mechanism to weld the payload and rewriter together to make self-rewriting possible. Here is an example of the use of `pastis-generator`. The payload here is a simple function which adds 42 to its parameter and the rewriter is the identity function.

```
(pastis-generator
 '((payload . (lambda (x) (+ 42 x)))
   (rewriter . (lambda (x) x))))
```

The resulting code-blend produced by the `pastis-generator` function is given below.

```
(lambda (args)
  (define (pastis-rewrite x)
    ((lambda (x) x) x))
  (define (pastis-payload x)
    ((lambda (x) (+ 42 x)) x))
  (define (pastis-ls l)
    (map (lambda (x) (write (pastis-rewrite x)) (display " ")) l))
  (define (pastis-code l)
    (display "(")
    (pastis-ls l)
    (display "(pastis-code '(")
    (pastis-ls l)
    (display ")) (pastis-payload args))\n"))
  (pastis-code
   '(lambda (args)
     (define (pastis-rewrite x)
       ((lambda (x) x) x))
     (define (pastis-payload x)
       ((lambda (x) (+ 42 x)) x))
     (define (pastis-ls l)
       (map (lambda (x) (write (pastis-rewrite x)) (display " ")) l))
     (define (pastis-code l)
       (display "(")
       (pastis-ls l)
       (display "(pastis-code '(")
       (pastis-ls l)
       (display ")) (pastis-payload\nargs))\n"))))
  (pastis-payload args))
```

5.2 Step by Step Explanations

The code generated by `pastis-generator` seems complicated, but its structure is in fact very similar to that of the following classical quine[4].

```
(define (d l) (map write-line l))
(define (code l) (d l) (display "(code '(\n") (d l) (display "))\n"))
(code '(
(define (d l) (map write-line l))
(define (code l) (d l) (display "(code '(\n") (d l) (display "))\n"))))
```

[4] A *quine* [8], named after Willard Van Orman Quine, is a program that prints its own code.

Adding a payload to this quine is quite straightforward.

```
(define (payload) (write "Hello, World!\n"))
(define (ls l) (map write-line l))
(define (code l) (ls l) (display "(code '(\n") (ls l) (display "))\n"))
(payload)
(code '(
  (define (payload) (write "Hello, World!\n"))
  (define (ls l) (map write-line l))
  (define (code l) (ls l) (display "(code '(\n") (ls l) (display "))\n"))
  (payload)))
```

Given Pastis's role, it is quite easy to see that it is related to quines. The only difference is that Pastis has to modify its code before printing it, instead of printing it *verbatim* as regular quines do. This is also quite easy to do.

However, deeper technical changes are required if we want to be able to pass parameters to the payload because the classical quine's structure does not permit this. The solution is to make a quine that is also a λ-expression (instead of a list of statements). This is possible, thanks to S-expressions.

The way the quine works relies on the fact that its code is a list of statements and that the last one can take a list of the previous ones as arguments. Making the whole quine a λ-expression in order to accept arguments for the payload means making it a single expression. But thanks to the language used, it appears that this single expression is *still* a list. This enables us to solve our problem. Here is the result:

```
(lambda (args)
  (define (payload x) (+ x 42))
  (define (ls l) (map write-line l))
  (define (code l)
    (display "(")
    (ls l) (display "(code '(")
    (ls l) (display "))\n(payload x))"))
  (code '(lambda (x)
           (define (payload x) (+ x 42))
           (define (ls l) (map write-line l))
           (define (code l)
             (display "(")
             (ls l) (display "(code '(")
             (ls l) (display "))\n(payload x))"))))
  (payload args))
```

5.3 Rewriter

In addition to `pastis-generator`, we also provide a `rewriter` function. Its role is to call specialized rewriters for each keyword, which will call rewriters recursively on their arguments if appropriate.

Specialized rewriters randomly choose a way to rewrite the top-level construct. In our example, the implemented rules are any interchange between `if`, `case` and `cond` (*i.e.* `if` $\leftrightsquigarrow$ `cond` $\leftrightsquigarrow$ `case` $\leftrightsquigarrow$ `if`) along with the transformation `if (condition) {A} else {B}` $\rightsquigarrow$ `if (!condition) {B} else {A}`. It is easy and trivial to change these rules.

5.4 Results

Pastis was tested with a simple payload and the example rewriter provided.

The code size increases steadily with generations, which seems to demonstrate that the `rewriter` function provided often adds new constructs, but seldom simplifies out the useless ones. As is clear from Pastis' structure, code size grows linearly as generations pass (right-hand graphic of Figure 7). In our experiment code size in megabytes seemed to grow as $\sim 15.35 \times$ generation.

The produced code is still fairly recognizable: keywords are not rewritten and highly specific intermediate variables appear everywhere in the code. Furthermore, the numerous tautological conditional branches (of the form `(if #t foo #!unspecific)`) and useless nesting of operators are also a sure way to identify code produced by Pastis. It is unclear if such artifacts could be used to conduct template power attacks to identify and remove polymorphic transformations. Given that a Lisp smart card does not exist to the best of that authors' knowledge, we could not practically test the effectiveness of this countermeasure *in vivo*.

Here is an example of the code produced by Pastis after some iterations.

```
(lambda (args) (define (pastis-rewrite x) ((lambda (x) (define
(rewrite-if s) (define get-cond (rewrite (cadr s))) (define
get-then (rewrite (caddr s))) (define get-else (let ((
key906856151243052050955551 38 (not (not (not (null? (cdddr s)))))))
(let ((key45066295344977747537902121 key906856151243052050955551 38
))(let ((key347530381576358560503334 13 (not (or (eq?
key45066295344977747537902121 (quote #f)))))) (let
((key748227697075550699293402 59 (not (or (eq?
key347530381576358560503334 13 (quote #f)))))) (cond ((not (not
(not (not (or (eq? key748227697075550699293402 59 (quote #t)))))))
(let ((key153009514049004536190920 96 #t)) (if (or (eq?
key153009514049004536190920 96 (quote #t))) (begin (case #t ((#t) (let
((key878388425855084564540664 7 (not (or (eq?
key748227697075550699293402 59 (quote #t)))))) (case (not (or (eq?
key878388425855084564540664 7 (quote #f))))((#t)(let ((
key417014702748854601217593 85 key878388425855084564540664 7))(if (not
(not (or (eq? key417014702748854601217593 85 (quote #t))))) (if (not
(or (eq? key417014702748854601217593 85 (quote #t)))) (let ((
key981341427931190418619807 07 #t)) (if (or (eq?
key981341427931190418619807 07 (quote #t))) (begin 42)
```

It is interesting to note that the self-referencing nature of Pastis makes it extremely hard to debug. When the third generation fails to run, for example, one needs to find the bug in the third generation code, identify what caused the second generation to produce it – and finally which code in the first generation caused the second generation to work this way. Several cases of bugs only occurring after several generations appeared during the development of Pastis.

Readers wishing to experiment with the three main program modules[5] can download them from [14].

5.5 Possible Extensions

The current `rewriter` function only serves as an example. First, it leaves several recognizable features in the code. More importantly, the transformations it applies are not very deep, since one could simply decide to only use `cond` constructs, systematically rewrite all `if` and `case` constructs to `cond` and focus on the rewriting of `cond`. To be more precise, `if` and `case` can be seen as Scheme syntactic sugar; it would probably be better to restrict the rewriting to a bare bones subset of the Scheme syntax, convert everything to this subset before rewriting and possibly convert some things back to syntactic sugar forms to make the rewritten code look more natural.

Several transformations could be applied instead of the simplistic operations done by our `rewriter` function. Here are a few ideas:

[5] `rewriters.scm`, `rewrite.scm` and `generator.scm`

α-Renaming The current `rewriter` does not rename variables at all. A way to do this would be to keep an environment indicating current renamings. When we encounter a definition, we change the name and add the original and rewritten name to the environment. When we encounter a name, we change it to the appropriate rewritten name by a simple lookup in the environment. It is assumed that when the Scheme virtual machine processes names, power signatures caused by processing different names will differ as well.

β-Reduction and β-Expansion A possible rewriting method would be to perform β-reductions (in the usual λ-calculus sense). Conversely, it would also be possible to perform β-expansions: select a sub-term, replace it by a variable and replace the whole expression by a function in this variable applied to the selected sub-term, taking all necessary care to prevent variable capture problems (roughly, ensuring that the operation does not bind other occurrences of the new variable and that the bindings of the sub-terms are still the same).

Of course, if we want to do such an operation without changing the semantics, we must ensure first that there is no breach of referential transparency in the code we are rewriting. Indeed, if side effects are taking place somewhere, the planned modifications could change the order of evaluation, or even the number of evaluations of some sub-terms.

Adding and Removing Definitions This would be the ability for the rewriter to add or remove local definitions when possible. When the rewriter sees an expression E(`expr`) it could replace it with `(let ((const expr))` E(`const`)`)`. This is very similar to the aforementioned β-reduction and expansion ideas and could be implemented in a similar way.

6 Avoiding Code Growth

While Pastis is conceptually interesting, the code growth problem makes Pastis useless for practical purposes. Let F_0 be the first generation of a self-rewriting program. Besides a payload representing the actual code's purpose, F_0 also contains a *non-deterministic* rewriting function H. H takes as input a version of the program and outputs another version, so that $\forall i \in \mathbb{N}$, $F_i = H(F_{i-1})$ with $i \neq j \Rightarrow F_i \neq F_j$ while retaining the code's core functionality, *i.e.* $F_i(m,k) = F_0(m,k)\ \forall\, i,m,k$, as shown in the left hand-side of Figure 7.

As in the basic Pastis example the size of F_i is monotonically increasing[6] (*i.e.*, $\text{size}(F_{i+1}) \geq \text{size}(F_i)$ with overwhelming probability), it is desirable to look for a different rewriting scheme[7].

[6] Code size is monotonically increasing *on the average*, we neglect the unlikely cases where rewriting will cause a decrease in code size.

[7] Note that it is theoretically impossible to require that both $i \neq j \Rightarrow F_i \neq F_j$ and $\forall i,\ \text{size}(F_i) <$ some bound.

An interesting alternative approach is to keep a representation of the original function F_0 within F_i and always rewrite F_0 differently. To ensure that each time a different program is created, the index i of the next version is passed to H: $F_i = H(F_0, i)$. Having F_i completely determined by F_0 and the index i can be helpful, especially for debugging purposes. However, this approach has a crucial drawback: An attacker may be able pre-compute all F_i and analyze them for side channel information leakage, thereby weakening the polymorphic protection. Thus, it is advisable to have an additional randomness source that makes H non-deterministic. If H is truly non-deterministic, there is no need to pass i as an argument to H because each call of $H(F_0)$ creates a new, randomized version of F_0. The left-hand side of Figure 8 illustrates such a system.

Note that at each rewritten generation, F_i is *completely* discarded and only the description of F_0 is used again for the new code generation. In addition, the description of F_0 does not need to be included in clear. If desired by the setting, it can instead be encrypted with a random key. For this, the corresponding encryption and decryption function as well as the random key, must be included with F_i and upon each code rewrite, the encrypted description of F_0 must be decrypted, rewritten by H, and encrypted again with a new random key. Interestingly, if the payload itself contains a block-cipher code, this code can be also used for encrypting F_0; thus no additional encryption routine needs to be embedded in the program. This encryption approach bears some similarities to techniques used by polymorphic viruses.

6.1 Separating H From F_i

Pastis is primarily meant for the protection of cryptographic functions from certain types of side-channel attacks. To this end, Pastis' primary goal is to rewrite the payload, while the rewriting process H itself is not directly subject to such attacks and thus may not need to be rewritten at all. This is because the functionality of H is independent of the public message input m and the secret input k.

Thus, it may appear fit for purpose to rewrite only F_0 and keep H intact. Such an approach is interesting as in many cases, H will be much more complex than the payload and may become even more complex (and maybe less efficient) after being rewritten, as rewriting rules can have a detrimental effect on the size and the efficiency of H. However, in some cases side channels emanating from H may leak information about the rewriting process and thus about the code of F_i. If this is the case, then the information gained from the attack could be used to subsequently create attacks on F_i. If, however, such an indirect attack on F_i is considered infeasible or unlikely for a particular instance of polymorphic code, then the approach of not rewriting H can be a practicable way to improve code efficiency. This may also allow for more complex rewriting rules that would not be possible if H had to be expressed in the possibly restricted realm of rewritable code (for example, the limited subset of Scheme used in Pastis).

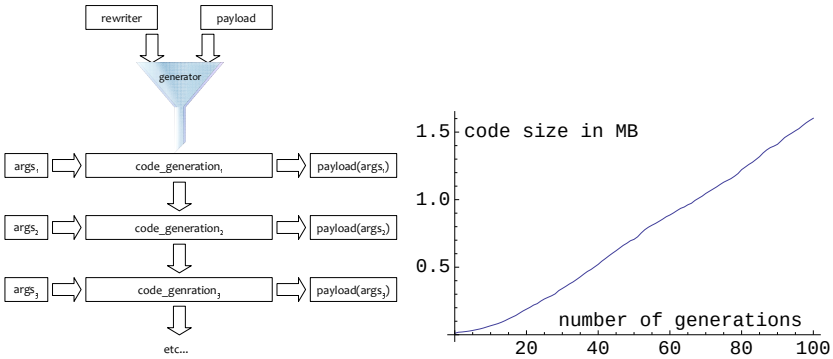

Fig. 7. Summary of the use of PASTIS (left) and typical evolution of code size with generations (right).

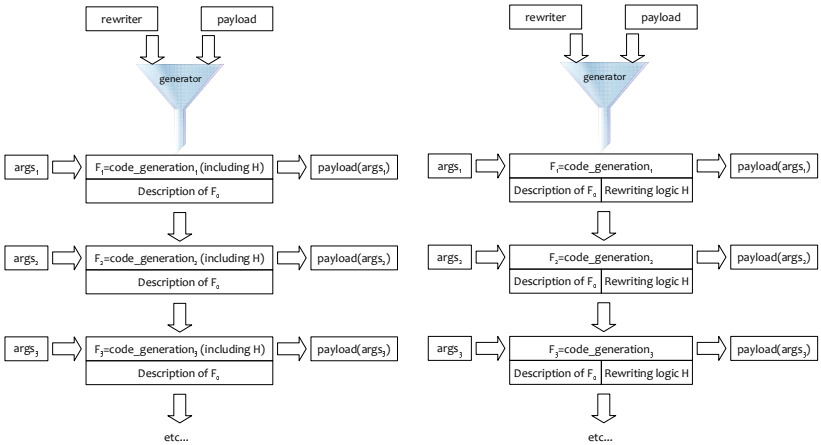

Fig. 8. Self-rewriting program without growth. Basic idea on the left, faster version on the right.

This motivates the suggestion of yet another modification of our paradigm: Instead of having H be a part of F_i, we may consider H and F_i as *separate* functions. On each invocation H is called first. H loads the description of F_0 and takes as additional (implicit) input either an index i and/or a source of randomness and outputs F_i, which – as above – has the same functionality as F_0 but is rewritten.

After rewriting F_0 as F_i, H passes the inputs $\{m, k\}$ to F_i to execute the desired payload. This is illustrated in the right-hand side of Figure 8.

6.2 Randomizing Compilers: A Practical Approach

It remains to decide how such a "description of F_0" that is included in each code generation should be chosen. From an implementor's perspective, the form of F_0's description must be chosen such that it consists only of elements that are understood by the rewriting engine while at the same time allows for a fast creation of F_i for all i. While it may appear natural to use the same type for F_0 as for F_i (*i.e.*, code that can directly be executed), using a more abstract representation has some advantages: For example, if a program is represented under the form of a syntax tree, it can be straightforward to analyze to find permutations of code blocks (*i.e.*, subtrees) that do not change the code's semantics. This is similar to our *buckets* idea in Section 2. When, in contrast, the program is represented as virtual machine code, such code rearrangements may be much more difficult to identify. Thus, F_0 should be described in a more abstract way and converted by H to a more concrete representation. In fact, we may consider H as a *compiler* that transforms code from a high-level language into a less abstract one.

For example, F_0 may be represented by a GIMPLE tree. GIMPLE [13] is a rather simple language-independent tree-representation of functions used extensively by the GNU Compiler Collection (GCC) in various stages of the compilation process. Source code from any language supported by GCC is transformed into GIMPLE which is then analyzed and optimized before being converted to the target language (for example, machine code).[8] Representing functions under GIMPLE removes much of the complexity from the compiler that would be needed when working directly with a high-level language like C++. This makes compilation very fast.

GCC applies many optimizations to GIMPLE trees which may change their form in several ways. This can be used to create very powerful polymorphic code: randomizing which of these optimizations are done and how exactly they are applied to the tree leads to many different possible results, all of which yield semantically equivalent code. Randomization can also be applied to the next compilation steps which turn the GIMPLE tree into the target language. As there are many ways to encode constructs like an `if` or simple arithmetic expressions

[8] This description of GCC's inner workings is – of course – greatly simplified.

into machine code, a great variety of possible realizations of such constructs can be found.

Thus, an extensive polymorphic framework can be built by using a *randomizing* version of the parts of GCC that deal with GIMPLE trees and transform them to machine code as H. Such a framework would allow the execution of very elaborate rewriting rules while preserving efficiency by only dealing with GIMPLE instead of source code.

The implementation of this approach is an idea that is yet to be explored – left as future work.

Acknowledgements

The work described in this paper has been supported in part by the European Commission IST Programme under Contract ICT-2007-216676 ECRYPT II and EPSRC grant EP/F039638/1.

References

1. G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti and S. Marchesin, Efficient Software Implementation of AES on 32-Bit Platforms, *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 2523, 159–171, 2002.
2. E. Brier, C. Clavier and F. Olivier, Correlation Power Analysis with a Leakage Model, CHES *2004*, Springer-Verlag LNCS 3156, 16–29, 2004.
3. C. Clavier, J.-S. Coron and N. Dabbous, Differential Power Analysis in the Presence of Hardware Countermeasures, *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 1965, 252–263, 2000.
4. C. Collberg and C. Thomborson, Watermarking, tamper-proofing, and obfuscation - tools for software protection, IEEE *Transactions on Software Engineering*, 28 (8), 735–746, 2002.
5. J. Daemen and V. Rijmen, *The Design of Rijndael*, Springer-Verlag, 2002.
6. C. Gentry, S. Halevi and V. Vaikuntanathan, i-Hop Homomorphic Encryption and Rerandomizable Yao Circuits, *Advances in Cryptology* CRYPTO *2010*, Springer-Verlag LNCS 6223, 155-172, 2010.
7. C. Herbst, E. Oswald and S. Mangard, An AES Smart Card Implementation Resistant to Power Analysis Attacks, *Applied Cryptography and Network Security (ACCS)*, Springer-Verlag LNCS 3989, 239–252, 2006.
8. D. Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid, Basic Books, (1999) [1979].
9. D. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*, Addison Wesley, 1998.
10. P. Kocher, J. Jaffe and B. Jun, Differential Power Analysis, *Advances in Cryptology* CRYPTO *1999*, Springer-Verlag LNCS 1666, 388–397, 1999.
11. P. Leadbitter, D. Page and N. Smart, Non-deterministic Multi-threading, IEEE *Transactions on Computers*, 56 (7), 992–998, 2007.

12. D. May, H. Muller and N. Smart, Non-deterministic Processors, *Australasian Conference on Information Security and Privacy (*ACISP*)*, Springer-Verlag LNCS 2119, 115–129, 2001.
13. J. Merrill, GENERIC and GIMPLE: A new tree representation for entire functions, Technical report, Red Hat, Inc. (2003) GCC Developer's Summit.
14. `http://pablo.rauzy.name/files/cryptomorph-sources.zip`
15. `http://www.gnu.org/software/mit-scheme/`
16. M. Tunstall and O. Benoît, Efficient Use of Random Delays in Embedded Software, *Information Security Theory and Practices (*WISTP*)*, Springer-Verlag LNCS 4462, 27–38, 2007.
17. Z. Xin, H. Chen, H. Han, B. Mao and L. Xie, Misleading Malware Similarities Analysis by Automatic Data Structure Obfuscation, ISC 2010, Information Security, Springer-Verlag LNCS 6531, 181–195, 2011.