Perturbated Functions: a new approach to Obfuscation and Diversity

Yoni De Mulder, Nessim Kisserli, Jan Cappaert, Nikos Mavrogiannopoulos, and Bart Preneel

Katholieke Universiteit Leuven
Dept. Electrical Engineering-ESAT/SCD/IBBT-COSIC
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
{firstname.lastname}@esat.kuleuven.be

Abstract. This paper presents a method for transforming a function into a combination of correlated perturbated functions by splitting its input space into sub-spaces, assigning each to a different function in such a way that a majority vote on their outputs will always yield the correct initial functions output. This novel technique allows function obfuscation and implementation diversication. Applications include software obfuscation, watermarking, and tamper resistance.

Keywords. Diversity, Software Security, Obfuscation, Perturbated Functions.

1 Introduction

The lack of diversity in our computing systems makes them more vulnerable to mass-attacks. Analysis of such incidents as the Morris worm [1], Code Red, Nimda, Blaster and Slammer concluded more diversity would have limited the breadth and speed of infection [2]. While Geer et al. [3] famously highlighted the security risk of monocultures, Schneider and Birman [4] pragmatically concluded that adding automated diversity to a deployed monoculture constituted a sensible defense.

The use of diversity in computing systems is not a new idea [5]. Its efficacy in protecting against failure is clearly valued in safety-critical systems such as aviation [6] where large sums of money are invested in using diverse primitives and processes, ranging from microchips, programming languages and compilers to the creation of independent software implementations from a common specification using N-version programming [5].

A related, but more economic approach to N-version programming is N-variant programming proposed by Evans et al. [7] in which synthetic diversity is used to create morphs of an input with distinct vulnerability sets designed to ensure no single input is able to break all variants. The morphs are executed in parallel, each only proceeding if all variants’ input-output behaviours agree.
Such an approach becomes more plausible as CPUs continue to incorporate ever more cores.

Despite numerous proposals for using diversity [8, 9] its deployment in commodity computing remains relatively modest. However, the recent incorporation of Address Space Layout Randomization (ASLR) [10] in most popular operating systems has raised the complexity of portable exploits [11, 12].

Another area that has yet to fully exploit diversification is license checking software. Crackers are still able to break a copy of some software, then produce patches which can be trivially applied by non-technical users to disable the license checking in their own copies. Clearly, destroying the viability of such small, easily distributed patches is desirable but by itself will not put an end to piracy. As higher bandwidth lowers the barriers to distributing whole-cracked binaries, other defense mechanisms are required to either identify the distributed binary or link it to a given execution environment.

The former is generally achieved through software fingerprinting, a form of software watermarking which relies on the indistinguishibility of the watermark from “normal” program code or data for tracing the origin. This makes diversified programs where differences among versions are inherent, natural candidates.

2 Perturbated Systems: the General Approach

2.1 Related work

Within the scope of traitor tracing, Billet et al. [13] proposed a traceable block cipher by providing many diversified descriptions of the same instance of the symmetric block cipher. The strength of the construction relies upon the Isomorphisms of Polynomials (IP) problem [14]. Unfortunately, Faugère et al. [15] presented an algorithm solving IP-like instances, hence defeating the approach of Billet et al. Following the introduction of internal perturbations to reinforce the IP-based cryptosystems [16], Bringer et al. reinforced the traceable block cipher [17]. Adding perturbations to the IP problem structure of each round of the traceable block cipher makes the algebraic structure less accessible to the adversary and hence protects against direct algebraic attacks. Bringer used the same idea in order to create a new white-box version of AES [18], i.e. a perturbated white-box AES implementation [19].

2.2 Perturbations and getting the correct value

Perturbations: The main goal of adding perturbations to an original system is to hide its actual structure such that it becomes less accessible to the adversary and hence less vulnerable to direct attacks. The perturbated system “often” gives the correct result, while in the other cases it just generates random values. More formally, the original system $S$ is replaced by the perturbated system $\tilde{S} = S + \tilde{0}$, where the perturbation system $\tilde{0}$ vanishes for some subset $X_s$ of the input space.


\[ \tilde{S}(x) = S(x) + \tilde{0}(x) = \begin{cases} 
S(x) & \text{if } x \in X_s \\
S(x) + r & \text{if } x \in \bar{X}_s 
\end{cases}, \quad (1) \]

where \( X = X_s \cup \bar{X}_s \) is the input space of \( S \) and \( r \) denotes a random value. While the original system \( S \) and the perturbation system \( \tilde{0} \) can be represented in many different ways, in [17, 19] these systems were represented as polynomials over a finite field \( \mathbb{F} \).

Get the correct value: Due to the introduction of the perturbations, there is a probability that the output of the perturbated system \( \tilde{S} \) is incorrect, more specifically random. In order to obtain always the correct output, we need four parallel perturbated systems \( \tilde{S}_i \mid i = 1, \ldots, 4 \) constructed using four correlated perturbation system \( \tilde{0}_i \mid i = 1, \ldots, 4 \) such that always two out of four systems \( \tilde{S}_i \) give the correct result while the other two result into different random values with a very high probability.

Being able to do so, we partition the input space \( X \) twice into two sets of approximately the same size, i.e. \( X = X_1 \cup \bar{X}_1 = X_2 \cup \bar{X}_2 \), and construct four correlated perturbation systems \( \tilde{0}_1, \tilde{0}_1, \tilde{0}_2 \) and \( \tilde{0}_2 \) accordingly such that the following is true for \( i = 1, 2 \):

\[ \forall x \in X_i : \tilde{0}_i(x) = 0 \quad \text{and} \quad \forall x \in \bar{X}_i : \tilde{0}_i(x) = 0 , \quad (2) \]
\[ \forall x \in \bar{X}_i : \tilde{0}_i(x) = r_1 \quad \text{and} \quad \forall x \in X_i : \tilde{0}_i(x) = r_2 , \quad (3) \]

where \( r_1, r_2 \) denote random values. With this construction, two out of four correlated perturbation systems vanish for any input \( x \) and hence the two corresponding perturbated systems return the correct value, whereas the other two output random values. Finally, a majority vote is used to filter out the correct result.

3 Perturbated Functions

In this section, we elaborate on how we apply the idea of perturbations discussed in Sect. 2 to \( n \)-to-\( m \)-bit functions \( f(\cdot) \), on how to randomize them, and on how these randomized perturbated functions can be implemented.

3.1 Generating perturbated functions \( f(\cdot)_{i,j} \)

Based on the general perturbation method mentioned in Sect. 2.2, a \( n \)-to-\( m \)-bit perturbated function \( \tilde{f}(\cdot)_{i,j} \) is constructed\(^1\) out of the original \( n \)-to-\( m \)-bit

\(^1\) The indexes \( i \) and \( j \) are explained in Sect. 3.2.
functions \( f(\cdot) \) as follows: generate a subset \( X_{i,j} \) of the input space \( X \) of \( f(\cdot) \) such that:

\[
\widetilde{f}(x)_{i,j} = \begin{cases} 
  f(x) & \text{if } x \in X_{i,j} \\
  r & \text{if } x \in X\setminus X_{i,j}
\end{cases}
\]

(4)

where \( r \) denotes a \( m \)-bit random value and \( X\setminus X_{i,j} \) is the complement of \( X_{i,j} \) with respect to \( X \), i.e. \( X\setminus X_{i,j} = X \setminus X_{i,j} \). In words, the perturbated function \( \widetilde{f}(x)_{i,j} \) only outputs the correct output value \( f(x) \) when the input \( x \) is an element of the chosen subset \( X_{i,j} \), and otherwise generates a random value independent of \( f(x) \).

### 3.2 Majority voting

As one can notice in (4), there is a probability that the perturbated function \( \widetilde{f}(x)_{i,j} \) gives back an incorrect random \( m \)-bit value \( r \), i.e. when \( x \in X\setminus X_{i,j} \). As mentioned in Sect. 2.2, we can always obtain the correct value \( f(x) \) by performing a **majority vote** on four parallel correlated perturbated functions \( \widetilde{f}(\cdot)_{i,j}|_{i=1,2;j=1,2} \) derived from the same original function \( f(\cdot) \). This requires the input space \( X \) of \( f(\cdot) \) to be partitioned twice into two sets of approximately the same size:

\[ X = X_{1,1} \cup X_{2,1} = X_{1,2} \cup X_{2,2} \]

with \( X_{2,j} = X_{1,j} = X \setminus X_{1,j} \) for \( j = 1, 2 \), and \( |X_{1,j}| \approx |X_{2,j}| \) for \( j = 1, 2 \). The partitioning can be seen as the creation of two pairs (indicated by index \( j \)) of complementary subsets (indicated by index \( i \)) of the same input space \( X \). For each of the four correlated subsets \( X_{i,j}|_{i=1,2;j=1,2} \) of \( X \), a perturbated function \( \widetilde{f}(\cdot)_{i,j} \) is constructed as defined in (4). By partitioning \( X \) differently and hence generating different subsets \( X_{i,j}|_{i=1,2;j=1,2} \), many different combinations of four correlated perturbated functions can be generated out of the same original function, which can be considered as a first degree of freedom.

Since each input \( x \) is always an element of exactly two out of four subsets \( X_{i,j}|_{i=1,2;j=1,2} \), i.e. exactly one subset \( X_{i,j} \) within each pair \( j = 1, 2 \), the corresponding two perturbated functions give back the correct value \( f(x) \) while the other two generate different random \( m \)-bit values with an overwhelming probability. This way, the four parallel correlated perturbated functions \( \widetilde{f}(\cdot)_{i,j}|_{i=1,2;j=1,2} \) can be seen as two pairs (indicated by \( j \)) each containing two perturbated functions \( \widetilde{f}(\cdot)_{i,j}|_{i=1,2;j=1,2} \) (indicated by \( i \)) of which exactly one of both generate the correct output \( f(x) \) for a given input \( x \). Hence the correct value \( f(x) \) can be selected by a majority vote (MV). Fig. 1 depicts an overview and a working example of a perturbated function where the input \( x \) is assumed to be an element of \( X_{1,1} \) and \( X_{2,2} \).

### 3.3 Encoded perturbated functions \( \widetilde{f}(\cdot)_{i,j}' \)

Until now, the intermediate results – defined as the output of the four parallel perturbated functions \( \widetilde{f}(x)_{i,j}|_{i=1,2;j=1,2} \) and accordingly the input of the majority voting – remain unencoded. For each possible input \( x \), two out of four
values \( \tilde{f}(x)_{i,j} |_{i=1,2; j=1,2} \) equal the correct value \( f(x) \) (the output of the original function), and hence these values are directly accessible by the adversary. Encoding these intermediate results masks the values \( \tilde{f}(x)_{i,j} |_{i=1,2; j=1,2} \) such that they become unaccessible. The encodings need to be annihilated at the input of the majority voting in order to guarantee its correct execution. Hereby, we force the attacker to perform a majority vote for each possible input \( x \) in order to obtain the corresponding original output \( f(x) \). Moreover, the introduction of encodings can be considered as a second degree of freedom, since there are many different equivalent encodings. Below, we first elaborate on the output encodings of the perturbated functions \( \tilde{f}(\cdot)_{i,j} |_{i=1,2; j=1,2} \), after which we explain how these encoded intermediate values can be interpreted correctly by the majority voting.

**Encoded perturbated functions** \( \tilde{f}^{\prime}(\cdot)_{i,j} \): the output of each perturbated function \( \tilde{f}(\cdot)_{i,j} \) is encoded/randomized by adding a random non-linear \( m \)-to-\((m+1)\)-bit output encoding \( g \). The combination of both results in the *encoded* perturbated function, denoted by \( \tilde{f}^{\prime}(\cdot)_{i,j} \), and is defined as follows:

\[
\tilde{f}(x)_{i,j}^{\prime} = g(f(x)_{i,j} \| j) = \begin{cases} 
g(f(x) \| j) & \text{if } x \in X_{i,j} 
g(r) & \text{if } x \in X_{i,j} \end{cases} ,
\]

(5)

where \( r \) denotes a \( m \)-bit random value. Below we discuss the properties of the output encoding \( g \) in detail:

- **\( m \)-to-(\( m+1 \))-bit**: The reason for a \( m \)-to-(\( m+1 \))-bit instead of a \( m \)-to-\( m \)-bit encoding \( g \) is the following: for each \( n \)-bit input \( x \), two out of four values \( f(x)_{i,j} |_{i=1,2; j=1,2} \) equal the correct \( m \)-bit value \( f(x) \), while the other
two produce different random \( m \)-bit values. To ensure that (i) all four values \( f(x)_{i,j} | i=1,2; j=1,2 \) are mapped to four different random encoded values\(^2\) \( f(x)_{i,j} | i=1,2; j=1,2 \) through \( g \) and (ii) those encoded values corresponding to \( f(x) \) can be remapped to \( f(x) \) through the inverse \( g^{-1} \) (see paragraph ‘Decoded majority voting’), we need two different random encoded values that map to the same \( m \)-bit value, and this for all possible \( m \)-bit values \( f(x) \). Hence this requires an extra bit, i.e. \( m+1 \);

- **Padding \( j \):** As can be observed in (5), the encoding \( g \) takes two inputs: (i) the output of the perturbated function, i.e. \( f(x)_{i,j} \), and (ii) the padding \( j \). Since two different random encoded \((m+1)\)-bit values are required for each \( m \)-bit value \( f(x) \), \( g \) acquires an additional input in order to decide to which one of both values it needs to map its \( m \)-bit input \( f(x)_{i,j} \). Since the four perturbated functions \( f(\cdot)_{i,j} | i=1,2; j=1,2 \) consist of two pairs \( j=1,2 \) each containing two perturbated functions \( f(\cdot)_{i,j} | i=1,2 \), and furthermore for each \( n \)-bit input \( x \) exactly one function in both pairs generate the correct \( m \)-bit output \( f(x) \), the additional input of \( g \) corresponds to the identifier of the pairs, i.e. \( j \).

The lack of the subindex \( i \) in the notation \( g \) indicates that the same random non-linear \( m \)-to-\((m+1)\)-bit output encoding is used for all four perturbated functions \( f(\cdot)_{i,j} | i=1,2; j=1,2 \).

**Decoded majority voting:** In the current setting, the output of the four encoded perturbated functions \( f(\cdot)_{i,j} | i=1,2; j=1,2 \) look completely random, unrelated to the original output \( f(x) \), but moreover all four are different with an overwhelming probability. This causes the majority voting (MV), as described in Sect. 3.2, to fail. Therefore, in order to guarantee MV’s correct execution, the output encoding \( g \) needs to be annihilated (inverted) at the input of MV. This is carried out by the non-linear \((m+1)\)-to-\( m \)-bit input decoding \( g^{-1} \), which remaps two out of four random encoded \((m+1)\)-bit values \( f(x)_{i,j} | i=1,2; j=1,2 \) back to the correct original \( m \)-bit value \( f(x) \) (exactly one for each pair \( j \)), while the other two are remapped to two different random \( m \)-bit values \( r_1 \) and \( r_2 \).

**Overview:** Fig. 2 depicts an overview and a working example of the random non-linear \( m \)-to-\((m+1)\)-bit output encoding \( g(\cdot|j) \) and the corresponding non-linear \((m+1)\)-to-\( m \)-bit input decoding \( g^{-1} \) where the input \( x \) is assumed to be an element of \( X_{1,1} \) and \( X_{2,2} \).

\(^2\) This ensure that the encoded outputs of \( f(\cdot)_{i,j} | i=1,2; j=1,2 \) look random, unrelated to \( f(x) \) and also independent of each other since all four are different.
3.4 Implementing encoded perturbated functions $\tilde{f}^{(\cdot)}_{i,j}$

There are various ways to implement the encoded perturbated functions $\tilde{f}^{(\cdot)}_{i,j}$, of which we discuss two in detail: lookup tables and interpolation polynomials.

**Lookup tables:** Inside a lookup table, the output values for all possible inputs are stored, where the input is used in order to index the table. Any function can be represented by lookup tables. More formally, given a function mapping a $n$-bit input to a $m$-bit output $f(x)$, the table stores the $m$-bit output for all possible $n$-bit inputs, hence the storage requirement is $2^n \cdot m$. In our case, applied to an encoded perturbated function $\tilde{f}^{(\cdot)}_{i,j}$ – as defined in (5) – mapping a $n$-bit input $x$ to a $(m + 1)$-bit output, the storage requirement would be:

$$2^n \cdot (m + 1).$$

(6)

As can be noticed in (6), the storage requirement of lookup tables is exponential in the input size $n$, hence lookup tables are only a practical solution for functions with a limited input size.

**Interpolation polynomials:** Another way to implement a function mapping a $n$-bit input $x$ to a $m$-bit output $f(x)$ is to use interpolation polynomials that will evaluate the correct $m$-bit value for every $n$-bit input. One of the possible ways to produce such a polynomial is by generating a *Lagrange form of the interpolating polynomial* $\mathcal{L}(x)$:

$$\mathcal{L}(x) = \sum_{l=1}^{2^n} \mathcal{L}_l(x),$$

where $\mathcal{L}_l(x) = f(x_l) \prod_{k=1; k \neq l}^{2^n} \frac{x - x_k}{x_l - x_k}$.
where the set \( \{ (x_i, f(x_i)) | i = 1, ..., 2^n \} \) contains all the data points that need to be emulated, assuming that all possible inputs (i.e. \( 2^n \)) and corresponding outputs are of importance. The degree of the polynomial is \( 2^n - 1 \).

This method has the advantage over the use of lookup tables of not requiring to represent all possible input and output values but rather only a polynomial that evaluates correctly on the required values. In our case, this advantage applies due to partitioning of the input space \( X \) of the original function \( f(\cdot) \) in two sets of approximately the same size, i.e. \( 2^n - 1 \) (half of the input space). As can be observed in (5), the encoded perturbated function \( \widetilde{f}(\cdot)_{i,j} \) generates a random value for inputs \( x \in X_{i,j} \). Hence, the interpolation polynomial for \( \widetilde{f}(\cdot)_{i,j} \) only needs to evaluate correctly for the data points \( (x, \widetilde{f}(x)_{i,j}) = (x, g(f(x)||j)) \) for \( x \in X_{i,j} \), where \( |X_{i,j}| \approx 2^{n-1} \).

Although \( \widetilde{f}(\cdot)_{i,j} \) maps \( n \)-bit values \( x \) to \((m+1)\)-bit values \( g(f(x)||j) \), in order to compute the storage requirement of the interpolation polynomial, we are assuming here that \( (m+1) = n \) and hence we have a mapping of \( n \) to \( n \) bits. In that special case, the polynomial is over the finite field \( GF(2^n) \) and needs to evaluate correctly for \( 2^{n-1} \) values. The storage requirement would be:

\[
2^{n-1} \cdot n ,
\]

since the Lagrange interpolation polynomial has at most \( 2^{n-1} \) coefficients of \( n \)-bit size.

**Comparison:** In the case when \( (m+1) = n \), implementing an encoded perturbated function \( \widetilde{f}(\cdot)_{i,j} \) as an interpolation polynomial in Lagrange form requires only half the storage space needed when implemented as a lookup table (comparing (6) with (7)). However, given an input \( x \), reading out an entry of a lookup table requires less computational effort than evaluating a polynomial of degree \( 2^{n-1} - 1 \).

**4 Applications**

In this section we show how this method could suit applications that require diversity and watermarking.

**4.1 Diversity**

The method discussed above has some advantages when it comes to diversity. In Sect. 3.2 and 3.3, we already pointed out two degrees of freedom: (i) constructing the perturbated functions \( \widetilde{f}(\cdot)_{i,j} |_{i=1,2; j=1,2} \) based on the partitioning of the input space \( X \) twice into two sets and (ii) the output encoding \( g \) are both flexible on their selection. For a given \( n \)-bit input \( x \), there are four different combinations in which two out of four perturbated functions \( \widetilde{f}(x)_{i,j} |_{i=1,2; j=1,2} \) give back
the correct output $f(x)$, i.e. two combinations within both pairs $j = 1, 2$ each containing two perturbated functions $\tilde{f}(x)_{i,j}|_{i=1,2}$ where exactly one results in $f(x)$. Repeating this for every $n$-bit input, the total number of different combinations of perturbated functions $\tilde{f}(\cdot)_{i,j}|_{i=1,2;j=1,2}$ due to different partitions of $X$ (regardless whether they are implemented using lookup tables or interpolation polynomials since the partitioning provides the diversity) is given by:

$$4(2^n) = 2(2^{n+1}) .$$

(8)

This combined with the choice of $g$, can provide additional diversity even at the output of the same function. Hence we can see that the perturbated function in general can provide a high degree of diversity at least to a class of pure functions that map $n$ to $m$ bits.

4.2 Watermarking

Software watermarking [20] is a way of embedding specific information into software. The main application of watermarking is its use to prove ownership of some intellectual property. In our scenario, intellectual property includes a secret software algorithm, embedded sensitive data, etc. Another application of watermarking is software fingerprinting. For example in a distribution of $n$ instances of a piece of software a unique signature is embedded in each instance. The embedded information binds a user to a specific software instance, allowing one to do “traitor tracing”. When an illegal software copy is discovered, the software owner can extract the watermark which will identify a certain user (the “traitor”), who is considered responsible for the abuse of its software instance.

The encoded perturbated functions are ideal for this information embedding. That is because the options for selection of $\tilde{f}(\cdot)_{i,j}|_{i=1,2;j=1,2}$, such as interpolation polynomials and lookup tables, allow for flexibility in the choice of their parameters that cannot be distinguished from a random choice. For example in a lookup table, the output values of selected invalid outputs could have a special meaning for the generator of the tables. Polynomials can also be used to carry extra information, such as passing through more given points, although the fact that the polynomial alone is not minimal might reveal that fact. However the extra points cannot be distinguished from normal input.

5 Conclusion and Future Work

We presented a method for transforming one function into many, diverse variants by splitting its input space into subspaces, assigning each to a different function in such a way that a majority vote on their outputs will always yield the correct initial function’s output. We also illustrated how the use of perturbations further obfuscates the functions while adding significant diversity to the implementation and highlighted some possible applications for the scheme.
These include targeted diversification of license checking code frustrating crackers’ ability to create simple portable patches for their cracks, and harnessing the individual diversity of each implementation to create a watermarking scheme.

As future work we consider the evaluation and efficiency of this approach in various setups. A first scenario is to use this as a the building blocks for a watermarking scheme. In addition to that, our approach can be evaluated as a diversifier in order to achieve (i) protection against code patching, and (ii) protection against code injection. To measure performance characteristics and impact on real world programs we plan to categorize which functions it can be applied to (e.g. pure functions in gcc) and evaluate it as a compiler plugin (e.g. in LLVM).

References

10. PaX Team: PaX address space layout randomization (ASLR)
11. Skylined: Internet Explorer IFRAME src and name parameter BoF remote compromise


