An Efficient Method for Random Delay Generation in Embedded Software*

Jean-Sébastien Coron and Ilya Kizhvatov

Université du Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg
{jean-sebastien.coron, ilya.kizhvatov}@uni.lu

Abstract. Random delays are a countermeasure against a range of side channel and fault attacks that is often implemented in embedded software. We propose a new method for generation of random delays and a criterion for measuring the efficiency of a random delay countermeasure. We implement this new method along with the existing ones on an 8-bit platform and mount practical side-channel attacks against the implementations. We show that the new method is significantly more secure in practice than the previously published solutions and also more lightweight.

Key words: Side channel attacks, countermeasures, random delays

1 Introduction

Insertion of random delays in the execution flow of a cryptographic algorithm is a simple yet rather effective countermeasure against side-channel and fault attacks. To our knowledge, random delays are widely used for protection of cryptographic implementations in embedded devices, mainly smart cards. It belongs to a group of *hiding* countermeasures, that introduce additional noise (either in time, amplitude or frequency domain) to the side channel leakage while not eliminating the informative signal itself. This is in contrary to *masking* countermeasures, that eliminate correlation between the side channel leakage and the sensitive data processed by an implementation.

Hiding countermeasures increase complexity of attacks while not rendering them completely impossible. They are not treated in academia as extensively as masking but are of great importance in industry. A mixture of multiple hiding and masking countermeasures would often be used in a real-life protected implementation to raise the complexity of attacks above the foreseen capabilities of an adversary.

There are two connected problems that arise in this field. The first one is to develop efficient countermeasures, and the second one is how to measure the efficiency of the countermeasures. In this paper we tackle both tasks for the case of the random delays.

* This is the updated version of the paper [5] from CHES 2009

Random delays. Most side-channel and fault attacks require an adversary to know precisely when the target operations occur in the execution flow. This enables her to synchronize multiple traces at the event of interest as in the case of Differential Power Analysis (DPA) and to inject some disturbance into the computations at the right time as in the case of fault attacks. By introducing random delays into the execution flow the synchronization is broken, which increases the attack complexity. This can be done in hardware with the so called *Random Process Interrupts* (RPI) as well as in software by placing "dummy" cycles at some points of the program. We give preliminary information on software random delays in Sect. 2.

Related work. First detailed treatment of the countermeasure was done by Clavier *et al.* in [4]. They showed that the number of traces for a successful DPA attack against RPI grows quadratically or linearly (when integration is used) with the standard deviation of the delay. Mangard presented statistical analysis of random disarrangement effectiveness in [7]. Amiel *et al.* [1] performed practical evaluation of random delays as a protection against fault attacks. Works of Bucci *et al.* [3] and Lu *et al.* [6] on random delays in hardware should also be mentioned here, the former suggesting the architecture for delay generation at the gate level, the latter implementing it on FPGA and addressing the optimization of delay generation parameters for this architecture.

To date, the only effort to *improve* the random delays countermeasure in software was published by Benoit and Tunstall in [9]. They suggested to modify the distribution of an individual independently generated random delay so that the variance of the sum increases and the mean, in turn, decreases. As a result, they achieve some improvement. We outline their method briefly here in Sect. 3.

Our Contribution. In this work, we propose a significantly more efficient algorithm for generating random delays in software (see Sect. 4). Our main idea is to generate random delays non-independently in order to obtain a much greater variance of the cumulative delay for the same mean.

We also introduce a method for estimating the efficiency of random delays based on the coefficient of variation (see Sects. 2 and 5). This method shows how much variance is introduced by the sum of the delays for a given performance overhead. We show that the plain uniform delays and the Benoit-Tunstall method [9] both have efficiency in $\Theta\left(1/\sqrt{N}\right)$ only, where N is the number of delays in the sum, whereas our method achieves $\Theta(1)$ efficiency with the growth of N. For example, compared to the plain uniform delays and to the Benoit-Tunstall method, for the sum of 10 delays our method is more than twice as efficient, and for the sum of 100 delays – over 6 times more efficient.

Finally, we implement our new method along with the previously known methods on an 8-bit Atmel AVR microcontroller and demonstrate by mounting practical side-channel attacks that it is indeed more efficient and secure (see Sect. 6). It is also more lightweight in terms of implementation.

2 Software Random Delays and Their Efficiency

A common way of implementing random delays in software is placing loops of "dummy" operations (like `NOP` instructions) at some points of the program. The number of loop iterations varies depending on the delay value.

A straightforward method is to generate individual delays independently with durations uniformly distributed in the interval $[0, a]$ for some $a \in \mathbb{N}$. We refer to this method as *plain uniform delays*. It is easily implementable in cryptographic devices as most of them have a hardware random number generator (RNG) on board.

In [4] and [7] it was shown that the complexity of a DPA attack (expressed as the number of power consumption traces required) grows quadratically or linearly (in case integration techniques are used) with the standard deviation of the trace displacement in the attacked point. That is why we are interested in making the variance of random delays as large as possible.

Here are our preliminary assumptions about the attacker's capabilities.

1. An attacker knows the times when the cryptographic algorithm execution starts and ends. This is commonly possible by monitoring I/O operations of a device, or operations like EEPROM access.
2. It is harder for an attacker to eliminate multiple random delays than a few ones.
3. The method of delay generation and its parameters are known to an attacker.

Note that it could be possible to place two sufficiently large and uniformly distributed delays in the beginning and in the end of the execution. That would make each point in the trace uniformly distributed over time when looking from the start of from the end, which is actually the worst case for an attacker. Unfortunately, in this case it would be relatively easy to synchronize the traces with the the help of cross-correlation (see [8] for an example). So we assume that in this case resynchronization of traces *can* be performed by an attacker. Therefore, we want to break the trace with relatively short (to keep performance) random delays in multiple places.

It can be still possible to detect delays produced by means of "dummy" loops in a side-channel trace because of a regular instruction pattern. To partially hinder this, "dummy" random data may be processed within a loop. We do not address this issue in this paper, just following the simple (but natural) assumption 2.

So an attacker will typically face the sum of several random delays. Following the Central Limit Theorem, the distribution of the sum of N *independent* (and *not necessarily uniform*) delays converges to normal with mean $N\mu_d$ and variance $N\sigma_d^2$, where μ_d and σ_d^2 are correspondingly the mean and the variance of the duration of an individual random delay. In other words, the distribution of the sum of independent delays depends only on the mean and the variance of individual delays but *not* on their particular distribution.

With all the above in mind, we adhere to the following criteria for random delay generation.

1. The sum of random delays from start or end to some point within the execution should have the greatest possible variance.
2. The performance overhead should be possibly minimal.

When estimating efficiency of random delay generation, one might be interested what performance overhead is required to achieve the given variation of the sum of N delays. Performance overhead can be naturally measured as the mean μ of this sum. We suggest to estimate efficiency of random delay generation methods in terms of the coefficient of variation σ/μ, where σ is the standard deviation for the sum of N random delays. The greater this efficiency ratio σ/μ, the more efficient the method is.

3 Method of Benoit and Tunstall

In [9], Benoit and Tunstall propose a way to improve the efficiency of the random delays countermeasure. Their aim is to increase the variance and decrease the mean of the sum of random delays while not spoiling the distribution of an individual random delay. To achieve this aim, the authors modify the distribution of an independently generated individual delay from the uniform to a pit-shaped one (see Figure 1). This increases the variance of the individual delay. Furthermore, some asymmetry is introduced to the pit in order to decrease the mean of an individual delay. The pit-shaped distribution is implemented by tabulating its inverse cumulative distribution function (c.d.f.).

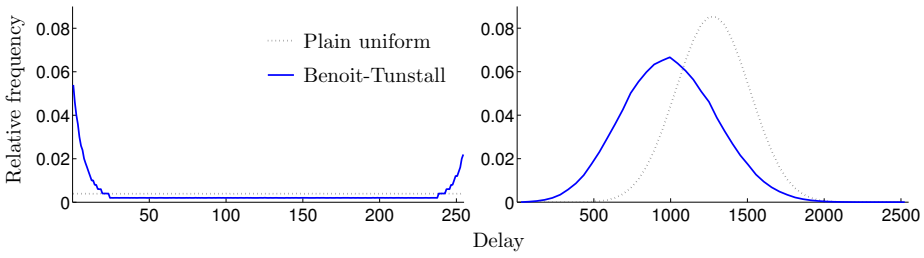

Fig. 1. Distribution for the method of Benoit and Tunstall [9] compared to plain uniform delays: 1 delay (left) and sum of 10 delays (right)

The delays are generated independently, so if an individual delay has mean μ_{BT} and variance σ^2_{BT}, the distribution of the sum of N delays converges to normal (as in the case of plain uniform delays) with mean $N\mu_{\mathrm{BT}}$ and variance $N\sigma^2_{\mathrm{BT}}$.

The authors estimate efficiency of their method by comparing it to plain uniform random delays. In an example, they report an increase of the variance by 33% along with a decrease of the mean by 20%. Distributions for a single delay and for the sum of 10 delays (for the parameters from the example mentioned above, see [9]) are shown in Figure 1 in comparison to plain uniform delays.

We note that the authors also pursued an additional criterion for the difficulty of deriving the distribution of the random delay. But it seems reasonable to consider this distribution to be known to an adversary, at least if the method is published.

4 Our New Method: Floating Mean

In this section we present our new method for random delay generation in software. The main idea of the method is to generate random delays non-independently. This significantly improves the variance of the cumulative delay and the method is also more efficient compared to [9] and to plain uniform random delays.

By $x \sim \mathcal{DU}[y, z]$ we will denote a random variable x following discrete uniform distribution on $[y, z]$, $y, z \in \mathbb{Z}$, $y < z$. We consider the parameters of the method below to be integers as in an embedded device integer arithmetic would be the only option when generating delays.

Our method is as follows. First, we fix some $a \in \mathbb{N}$ which is the maximum delay length. Additionally, we fix another parameter $b \in \mathbb{N}$, $b \leq a$. These implementation parameters a and b are fixed in an implementation and do not change between different executions of an algorithm under protection.

Now, in each execution, we first produce a value $m \in \mathbb{N}$ randomly uniformly on $[0, a-b]$, and then generate individual delays independently and uniformly on $[m, m+b]$. In other words, within any given execution individual random delays have a fixed mean $m + b/2$. But this mean varies from execution to execution, hence our naming of the method.

The resulting histograms in comparison to plain uniform delays are depicted in Figure 2. This figure also shows how the properties of the method vary dependent on the ratio b/a of the parameters of the method, that can take possible values between 0 and 1.

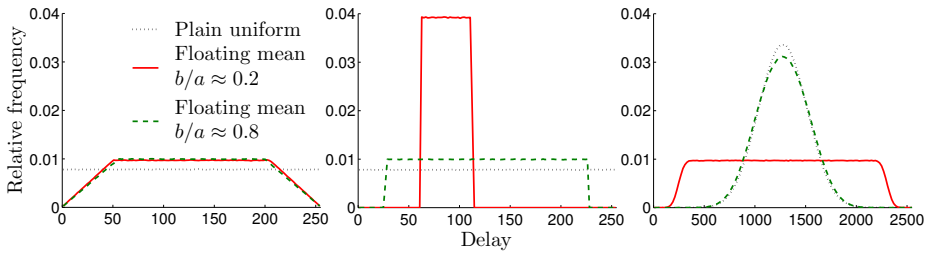

Fig. 2. Distribution for the Floating mean method with different b/a ratio compared to plain uniform delays: histogram for 1 delay (left), for 1 delay within a single trace, *i.e.* for some fixed m (center) and for the sum of 10 delays (right), $a = 255$

In fact, Floating mean is a pure trade-off between the quality of the distribution of single delay within a trace and that of the sum of the delays. When b/a is small (like the case $b = 50$, $a = 255$, $b/a \approx 0.2$ in Figure 2), the distribution of an individual delay within a trace has a comparatively small variance, but the variance of a single delay across traces and of the sum of the delays is large. When b/a is large (like the case $b = 200$, $a = 255$, $b/a \approx 0.8$ in Figure 2), the distribution of an individual delay within a trace has large variance, but the distribution of the sum of the delays converges to normal. The extreme case $b/a = 0$ just means that within an execution all delays have same length m, while the distribution of the sum of N delays is uniform on the N-multiples in $[0, aN]$. In the other extreme case, $b/a = 1$, the methods simply converges to plain uniform delays with each delay generated uniformly on $[0, a]$.

To calculate the parameters of the distribution of the sum S_N of N delays, we represent an individual delay as a random variable $d_i = m + v_i$, where $m \sim \mathcal{DU}[0, a - b]$ and $v_i \sim \mathcal{DU}[0, b]$ for $i = 1, 2, \ldots N$ are independent random variables. The sum is then expressed as

$$S_N = \sum_{i=1}^{N} d_i = Nm + \sum_{i=1}^{N} v_i \, .$$

For the mean, we have

$$E(S_N) = E(Nm) + E\left(\sum_{i=1}^{N} v_i\right) = N \cdot \frac{a-b}{2} + N \cdot \frac{b}{2} = \frac{Na}{2} \, .$$

For the variance, since m and v_i are independent, all v_i are identically distributed and

$$\mathrm{Var}(m) = \frac{(a-b+1)^2 - 1}{12} \, , \quad \mathrm{Var}(v_i) = \frac{(b+1)^2 - 1}{12} \, , \quad i = 1, 2, \ldots, N$$

we have

$$\begin{aligned} \mathrm{Var}(S_N) &= \mathrm{Var}\left(Nm + \sum_{i=1}^{N} v_i\right) = N^2 \cdot \mathrm{Var}(m) + N \cdot \mathrm{Var}(v_1) \\ &= N^2 \cdot \frac{(a-b+1)^2 - 1}{12} + N \cdot \frac{b^2 + 2b}{12} \, . \end{aligned}$$

So, the variance of the sum of N delays is in $\Theta\left(N^2\right)$, in comparison to plain uniform delays and the method of [9] that both have variances in $\Theta\left(N\right)$. This is because we generate random delays non-independently; namely in our solution the lengths of the individual random delays are correlated: they are short if m is small, or they are longer if m is larger. This enables us to get a much larger variance than if the delays were generated independently, as in the plain uniform method and the method of [9].

At the same time, if we look at the delays within a single execution and thus under fixed m, the mean for the sum of N delays becomes $N(m + b/2)$. This implies that the cumulative delay for a given execution and therefore the length of the execution depends on m. An adversary can thus accept only the short traces, as they have short individual delays, and reject the long ones; this can lower the complexity of the attack.

In order to relieve an adversary of such a benefit, we can generate the first half of random delays (in the first half of the execution) uniformly on $[m, m+b]$ (that is, with mean $m + b/2$), and the second half of delays – uniformly on $[a-m-b, a-m]$ (that is, with mean $a-m-b/2$). In this way, the distribution of the sum of all the $N = 2M$ delays for a given execution is independent of m (the mean is $aN/2$ and the variance is $N(b^2 + 2b)/12$). So an adversary cannot gain any additional information about the distribution of the delays within an execution by observing its length. Still, the variance of the sum of $L < M$ delays from start or end to some point up to the middle of the execution is in $\Theta(L^2)$.

Floating mean method is described in Algorithm 1.1. It is easily implementable in software on a constrained platform that has a built-in RNG producing uniformly distributed bytes since parameters a and b can be naturally chosen so that $a - b = 2^s - 1$ and $b = 2^t - 1$, where $s, t \in \mathbb{N}$ and $2^s + 2^t < 2^n + 2$ for an n-bit target microcontroller. Random integers in the range $[0, 2^s - 1]$ and $[0, 2^t - 1]$ can be obtained by a simple bit-wise `AND` with bit masks $2^s - 1$ and $2^t - 1$ correspondingly. The method requires no additional memory, as opposed to [9]. We are describing our implementation of Floating mean in Sect. 6 and Appendix B.

Algorithm 1.1 Floating mean method for generation of random delays

Input: $a, b \in \mathbb{N}, b \leq a, N = 2M \in \mathbb{N}$
  $m \leftarrow \mathcal{DU}[0, a - b]$
  for $i = 1$ to $N/2$ **do**
    $d_i \leftarrow m + \mathcal{DU}[0, b]$
  end for
  for $i = N/2 + 1$ to N **do**
    $d_i \leftarrow a - m - \mathcal{DU}[0, b]$
  end for
Output: $d_1, d_2, \ldots, d_N$

4.1 A Method That Does Not Quite Work: Floating Ceiling

In this section we present another method that is based on the same principle as the previous method: generate random delays non-independently to improve the variance of the cumulative sum. However we explain below why this method does not quite work.

The method is as follows. First, we fix some implementation parameter $a \in \mathbb{N}$ which determines the maximum length of an individual random delay. Now, prior

to generation of the first delay in each execution of the algorithm we produce a value $c \in \mathbb{N}$ randomly uniformly on $[1, a-1]$. After that, within the execution we generate individual delays randomly uniformly on $[0, c]$. Loosely speaking, c is the "ceiling" for the length of the random delays that varies from execution to execution. The resulting distributions are shown in Figure 6 in Appendix A. For the sum S_N of N delays we obtain the following mean and variance (see Appendix A):

$$E(S_N) = N \cdot \frac{a}{4}\,, \quad \mathrm{Var}(S_N) = N^2 \cdot \frac{a^2 - 2a}{48} + N \cdot \frac{2a^2 + 5a}{72}\,.$$

As in the Floating mean method, here the variance of the sum of the delays is also in N^2 since we generate delays non-independently. However we have the same undesired property as in the Floating mean method without the two halves. Namely the mean length of the cumulative delay within a single trace (*i.e.* with c fixed) is $Nc/2$. So an adversary can judge the mean length of the delays within an execution by the total length of the execution that he can definitely measure.

If we try to fix this in the same manner by generating the first half of random delays uniformly on $[0, c]$ and the second half – uniformly on $[0, a-c]$, the mean of the sum of all $N = 2M$ random delays within an execution becomes constant and equal to $Na/4$. However, one can see that for a given execution the distribution of the sum (and in particular its variance) still depends on c; therefore an adversary could still derive information from c in a given execution by measuring its length. For example, since the variance of the sum is maximal when $c = 0$ or $c = a$, an adversary could select those executions in which a large deviation from the mean is observed; this would likely correspond to small c or large c; then the adversary would concentrate his attack on those executions only.

The complete Floating ceiling method is defined by Algorithm 1.2. It does not require any tables to be stored in memory, as opposed to [9]. However, its implementation requires random integers on $[0, c]$ for arbitrary positive integer c. This can be inconvenient on constrained platforms as this requires to omit RNG outputs larger than c, thus leading to a performance decrease.

Algorithm 1.2 Floating ceiling method for generation of random delays

Input: $a \in \mathbb{N}, N = 2M \in \mathbb{N}$
  $c \leftarrow \mathcal{DU}[1, a-1]$
  for $i = 1$ to $N/2$ **do**
    $d_i \leftarrow \mathcal{DU}[0, c]$
  end for
  for $i = N/2 + 1$ to N **do**
    $d_i \leftarrow \mathcal{DU}[0, a-c]$
  end for
Output: $d_1, d_2, \ldots, d_N$

5 Comparing Efficiency

In this section we compare our new method with the existing ones based on the efficiency metrics σ/μ suggested in Sect. 2.

Efficiency ratios σ/μ for the sum of N delays for the new method and for the existing ones are given in Table 1. Note that we are mostly interested in the coefficient of variation somewhere around the middle of the trace.

Table 1. Efficiency ratios σ/μ for different random delay generation methods

Plain uniform	Benoit-Tunstall	Floating mean
$\frac{1}{\sqrt{3N}} = \Theta\left(\frac{1}{\sqrt{N}}\right)$	$\frac{\sigma_{\mathrm{BT}}}{\mu_{\mathrm{BT}}} \cdot \frac{1}{\sqrt{N}} = \Theta\left(\frac{1}{\sqrt{N}}\right)$	$\frac{\sqrt{N((a-b+1)^2-1)+b^2+2b}}{a\sqrt{3N}} = \Theta(1)$

In Figure 3, the efficiency ratio σ/μ for the sum of N delays for different methods is depicted against N. For all methods, we have considered the maximum delay length $a = 255$. The mean $\mu_{\mathrm{BT}} = 99$ and the variance $\sigma^2_{\mathrm{BT}} = 9281$ of an individual delay in the Benoit-Tunstall method was estimated empirically for the parameters used as an example in [9].

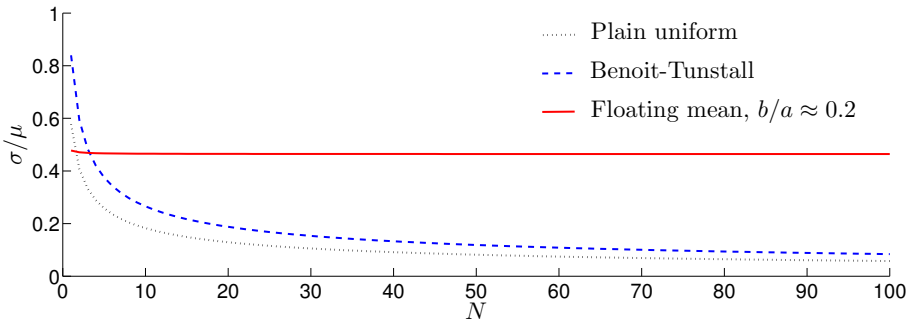

Fig. 3. Efficiency of the random delay generation algorithms in terms of the efficiency ratio σ/μ against the number of delays N in a sum

It can be seen that our new Floating mean method presented in Sect. 4 is more efficient compared to the previously published ones. Figure 4 further illustrates the difference, presenting the distributions of the sum of 100 random delays for different methods with the parameters that yield the same performance penalty, *i.e.* the same mean of the sum. We see that for the same average performance penalty, our method has a much larger variance.

In the case of independently generated individual delays the efficiency ratio σ/μ for the sum of any N delays is $\sigma_d/\mu_d \cdot 1/\sqrt{N}$, where σ_d and μ_d are the stan-

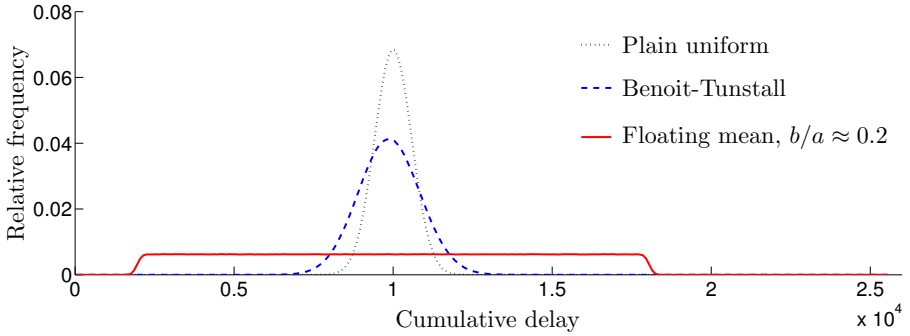

Fig. 4. Distributions of the sum of 100 delays for random delay generation algorithms, for the case of equal means

dard deviation and the mean of an individual delay. One can increase σ_d/μ_d ratio to improve efficiency, which was done in [9], but with an increase of the number of delays in the sum the efficiency of such methods decreases like $\Theta\left(1/\sqrt{N}\right)$, asymptotically tending to 0. Whereas for our method the efficiency is in $\Theta(1)$, so with an increase of the number of delays it tends to a nonzero constant value. This can be seen in Figure 3.

Thus, when implementing our method, one can benefit from using shorter but more frequent delays, as this does not cause the decrease in efficiency. This is an advantage as frequent short delays may be harder to eliminate than the long but less frequent ones.

6 Implementation and Resistance to Practical Attacks

Here we present comparison between the practical implementations of plain uniform delays, table method of Benoit and Tunstall [9] and the new Floating mean method by mounting Correlation Power Analysis (CPA) attack [2] against them.

We have implemented the methods on an 8-bit Atmel AVR microcontroller. Each delay is a multiple of 3 processor cycles (this granularity cannot be further reduced for this platform). Further details on our implementation are presented in Appendix B.

Random delays were introduced into AES-128 encryption. We put 10 delays per each round: before `AddRoundKey`, 4 per `SubBytes`+`ShiftRows`, before each `MixColumn` and after `MixColumns`. 3 "dummy" AES rounds that also incorporated random delays were added in the beginning and in the end of the encryption. Thus, the first `SubByte` operation of the first encryption round, which is the target for our attacks, is separated from the start of the execution, which is in turn our synchronization point, by 32 random delays.

The parameters of the methods were chosen to ensure (nearly) the same performance overhead across the methods. They were made sufficiently small

to enable attacks with a reasonable number of traces. For the Floating mean method we used parameters $a = 18$ and $b = 3$. For the table method of Benoit and Tunstall, the p.d.f. of the pit-shaped distribution was generated using the formula $y = \lceil ak^x + bk^{N-x} \rceil$ from [9] with the parameters $N = 19$, $a = 40$, $b = 34$ and $k = 0.7$. These parameters were chosen so that they lead to the table of 256 entries with the inverse c.d.f. of the distribution (see Table 3 in Appendix B). We use this table to produce delay values on $[0, 19]$ by indexing it with a random byte. For the plain uniform delays, the individual delay values were generated on $[0, 16]$. On our 8-bit platform we can efficiently produce random integers only on $[0, 2^i - 1]$ for $i = 1, 2, ..., 8$ (see Sect. 4), so we could not make the performance overhead for this method to be exactly the same as for the other methods.

We mounted CPA attack [2] in the Hamming weight power consumption model against the first AES key byte for each of the methods, first `SubByte` operation being the attack target. As a reference benchmark for our measurement conditions we performed CPA attack against the implementation without random delays. For implementations with random delays, we used power consumption traces *as is* without any alignment or integration to make a consistent comparison. Table 2 presents the number T_{CPA} of traces required for a successful (with the 1st-order success rate close to 1) key byte recovery along with estimated mean μ, standard deviation σ and efficiency ratio σ/μ of the sum of 32 delays for each of the methods. Figure 5 presents the CPA attack results.

Table 2. Practical effect of the sum of 32 delays for different methods

	No delays	Plain uniform	Benoit-Tunstall [9]	Floating mean
μ, cycles	0	720	860	862
σ, cycles	0	79	129	442
σ/μ	–	0.11	0.15	0.51
T_{CPA}, traces	50	2500	7000	45000

It can be seen that the Floating mean method is more secure in practice already for small delay durations and for a small number of delays. To break our implementation, we require 45000 traces for Floating mean and 7000 traces for Benoit-Tunstall. That is, for the *same performance penalty* the Floating mean method requires 6 times more curves to be broken. This ratio will increase with the number of delays. However, our method is more efficient already for less than 10 delays in the sum, as can be seen from Figure 4. This is important for symmetric algorithm implementations that are relatively short. For inherently long implementations of public key algorithms the number of delays in the sum will be naturally large.

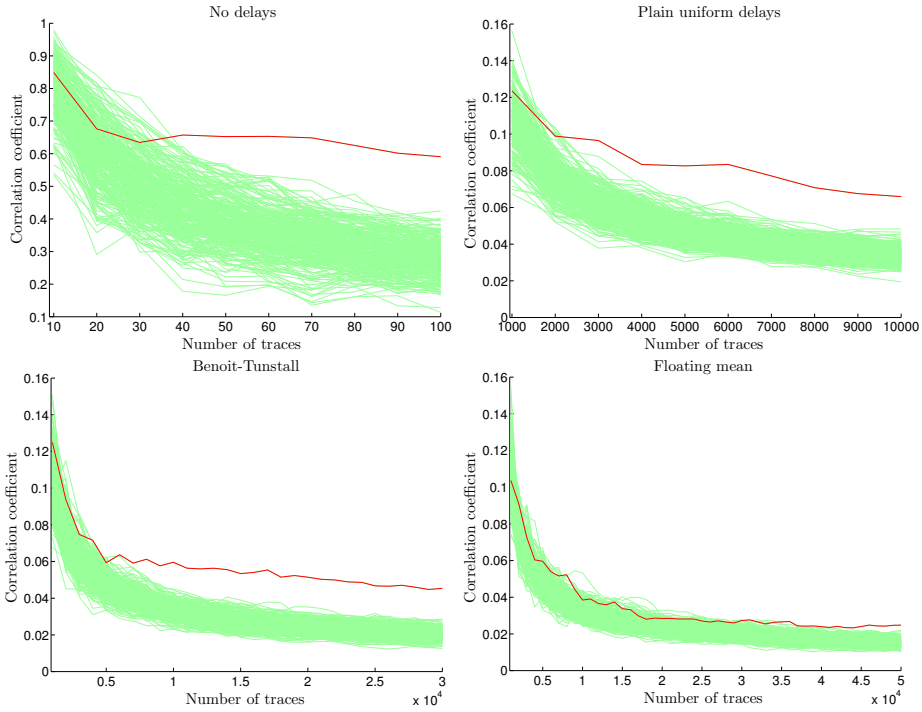

Fig. 5. CPA against random delays: correlation coefficient for all key byte guesses against the number of power consumption traces. The trace for the correct guess is highlighted.

7 Conclusion

We proposed a new method for random delay generation in embedded software – the Floating mean method – and introduced a way to estimate efficiency of the random delays countermeasure. We presented the lightweight implementation of our method for protection of AES encryption on an 8-bit platform. We mounted practical CPA attacks showing that for the same level of performance the reference implementation of the new method requires 6 times more curves to be broken compared to the method of Benoit and Tunstall [9]. Thus, our method is significantly more efficient and secure.

References

1. Frederic Amiel, Christophe Clavier, and Michael Tunstall. Fault analysis of DPA-resistant algorithms. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Paul Seifert, editors, FDTC *2006*, volume 4236 of LNCS, pages 223–236. Springer, Heidelberg, 2006.
2. Eric Brier, Christophe Clavier, and Olivier Benoit. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, CHES *2004*, volume 3156 of LNCS, pages 135–152. Springer, Heidelberg, 2004.
3. Marco Bucci, Raimondo Luzzi, Michele Guglielmo, and Alessandro Trifiletti. A countermeasure against differential power analysis based on random delay insertion. In IEEE *International Symposium on Circuits and Systems* – ISCAS *2005*, volume 4, pages 3547–3550, May 2005.
4. Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential power analysis in the presence of hardware countermeasures. In Çetin Kaya Koç and Christof Paar, editors, CHES *2000*, volume 1965 of LNCS, pages 252–263. Springer, Heidelberg, 2000.
5. Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Christophe Clavier and Kris Gaj, editors, CHES *2009*, volume 5747 of LNCS, pages 156–170. Springer, Heidelberg, 2009.
6. Yingxi Lu, Maire P. O'Neill, and John V. McCanny. FPGA implementation and analysis of random delay insertion countermeasure against DPA. In *International Conference on Field-Programmable Technology* – FPT *2008*, pages 201–208, 2008.
7. Stefan Mangard. Hardware countermeasures against DPA – a statistical analysis of their effectiveness. In Tatsuaki Okamoto, editor, CT-RSA *2004*, volume 2964 of LNCS, pages 222–235. Springer, Heidelberg, 2004.
8. Sei Nagashima, Naofumi Homma, Yuichi Imai, Takafumi Aoki, and Akashi Satoh. DPA using phase-based waveform matching against random-delay countermeasure. In IEEE *International Symposium on Circuits and Systems* – ISCAS *2007*, pages 1807–1810, May 2007.
9. Michael Tunstall and Olivier Benoit. Efficient use of random delays in embedded software. In Damien Sauveron, Konstantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater, editors, WISTP *2007*, volume 4462 of LNCS, pages 27–38. Springer, Heidelberg, 2007.

A Distribution for the Floating Ceiling Method

To calculate the mean and the variance for the Floating ceiling method, we represent an i-th individual delay as a random variable $d_i \sim \mathcal{DU}[0, c]$, $i = 1, 2, \ldots N$, where in turn $c \sim \mathcal{DU}[1, a-1]$. The sum of N delays is expressed as

$$S_N = \sum_{i=1}^{N} d_i \, .$$

For the mean, since d_i are identically distributed, we have

$$E(S_N) = NE(d_1) = N \sum_{c=1}^{a-1} \frac{1}{a-1} E\left(d_1 | c\right) = N \cdot \frac{1}{a-1} \sum_{c=1}^{a-1} \frac{c}{2} = N \cdot \frac{a}{4} \, .$$

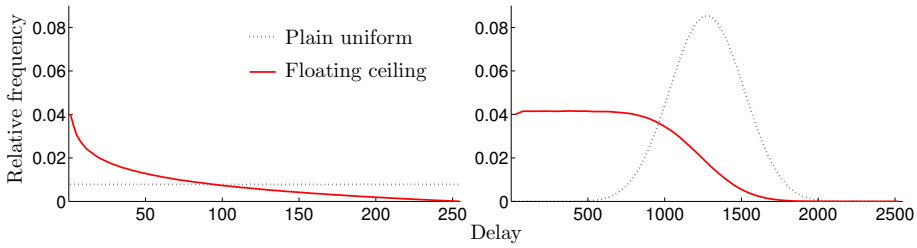

Fig. 6. Distribution for the Floating ceiling compared to plain uniform delays: 1 delay (left) and sum of 10 delays (right) for $a = 255$

For the variance, in turn,

$$\mathrm{Var}(S_N) = E\left(S_N^2\right) - \left(E\left(S_N\right)\right)^2 \ .$$

Again, since d_i are identically distributed, we have

$$E\left(S_N^2\right) = E\left(\left(\sum_{i=1}^{N} d_i\right)^2\right) = E\left(\sum_{i=1}^{N} d_i^2\right) + 2E\left(d_1d_2 + d_1d_3 + \ldots + d_{N-1}d_N\right)$$
$$= NE\left(d_1^2\right) + \binom{N}{2} \cdot 2E\left(d_1d_2\right) \ .$$

Now, having

$$E\left(d_1^2\right) = \sum_{c=1}^{a-1} \frac{1}{a-1} E\left(d_i^2 \mid c\right) = \frac{1}{a-1} \sum_{c=1}^{a-1} \frac{1}{c+1} \sum_{j=0}^{c} j^2 = \frac{4a^2+a}{36}$$

and (since $d_i|c$ and $d_j|c$ are independent for $i \neq j$ and identically distributed)

$$E\left(d_1d_2\right) = \sum_{c=1}^{a-1} \frac{1}{a-1} E\left(d_1d_2 \mid c\right) = \frac{1}{a-1} \sum_{c=1}^{a-1} \left(E(d_1 \mid c)\right)^2 = \frac{2a^2-a}{24} \ ,$$

we finally obtain

$$\mathrm{Var}(S_N) = N^2 \cdot \frac{a^2-2a}{48} + N \cdot \frac{2a^2+5a}{72} \ .$$

B Implementation of Random Delays for an 8-bit AVR Platform

Here we present the reference implementation of several delay generation methods in the 8-bit AVR assembly language. Throughout the code, the following registers are reserved: `RND` for obtaining the random delay duration, `FM` for storing

the value of m used in Floating mean during the execution, `MASK` for the bit mask that truncates random values to the desired length.

Common ATmega16 microcontroller that we used does not have a built-in RNG. Hence, we have simulated the RNG by pre-loading a pool of pseudorandom numbers to microcontroller's SRAM from the host PC prior to each execution and pointing the `X` register at the beginning of the pool. Random numbers are then loaded successively from SRAM to `RND` register by calling the `randombyte` function:

```
randombyte:
    ld  RND, X+  ; X is the dedicated address register
    ret          ;  that is used only in this function
```

First, here is the basic delay generation routine. It produces delays of length $3 \cdot$ `RND`$+C$ cycles, where C is the constant overhead per delay. To reduce this overhead, the delay generation can be implemented in-line to avoid the cost of entering and leaving the function. The part of the code specific for delay generation methods is omitted and will be given below.

```
randomdelay:
    rcall randombyte  ; obtain a random byte in RND
    ;
    ; <place for method-specific code>
    ;
    tst   RND    ; mind balancing between zero and
    breq  zero   ;  non-zero delay values!
    nop
    nop
dummyloop:
    dec   RND
    brne  dummyloop
zero:
    ret
```

Here are specific code parts for delay value generation. For plain uniform delays, the code is just:

```
    and   RND, MASK     ; truncate random value to the desired length
```

The code for the floating mean is one instruction longer (namely, addition of the value m).

```
    and   RND, MASK     ; truncate random value to the desired length
    add   RND, FM       ; add 'floating mean'
```

Floating mean also requires initialization (namely, generation of m) in the beginning of each execution:

```
    rcall randombyte    ; obtain a random byte in RND
    mov FM, RND
    ldi MASK, 0x0f
    and FM, MASK        ; trucate mean to the desired length
    ldi MASK, 0x03      ; set mask for future use in individual delays
```

and "flipping" `FM` in the middle of the execution to make the total execution length independent of the value of m.

```
ldi MASK, 0x0f
sub MASK, FM
mov FM, MASK
ldi MASK, 0x03
```

Finally, for the method of Benoit and Tunstall, the delay value is generated as follows.

```
ldi ZH, high(bttable)
mov ZL, RND
ld RND, Z
```

Here `bttable` is the table of 256 byte entries with the inverse c.d.f. of the pit-shaped distribution that is pre-loaded into SRAM. The table used in our experiments reported in Sec. 6 is given in Table 3.

Table 3. Tabulated inverse c.d.f. of the pit-shaped distribution (hexadecimal notation)

```
00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,
00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,01,01,01,01,01,01,01,01,01,01,
01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,02,02,02,02,
02,02,02,02,02,02,02,02,02,02,02,02,02,02,02,02,02,02,03,03,03,03,03,03,03,03,
03,03,03,03,03,03,03,03,04,04,04,04,04,04,04,04,04,04,05,05,05,05,05,05,05,06,
06,06,06,06,06,06,07,07,07,07,08,08,08,09,09,09,0a,0a,0a,0b,0b,0b,0c,0c,0c,0c,
0d,0d,0d,0d,0d,0e,0e,0e,0e,0e,0e,0f,0f,0f,0f,0f,0f,0f,0f,0f,10,10,10,10,10,10,
10,10,10,10,10,10,10,10,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,12,12,
12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,
12,12,12,12,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,
13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13
```

It can be seen that Floating mean is more "lightweight" in terms of both memory and code than the table method of Benoit and Tunstall.