Practical Aspects of PUFs

PUFs are going commercial as we speak (as of the writing of this on 3/2013)

In their move from academic/theoretical exercises to commercial products, the **practical** aspects of the PUF need to be addressed

We already discussed (in the statistics lecture) several metrics on which PUFs are evaluated

- Randomness and Bias (NIST tests)
- Uniqueness (Average *inter-chip* Hamming Distance, and their distribution characteristics)
- Stability (Average *intra-chip* Hamming Distance)

Some metrics are more important than others, depending on the **usage scenario**

Practical Aspects of PUFs

Usage Scenarios:

• The PUF will be used as a 'serial number' to identify and/or track parts through manufacturing or in the field

This usage scenario has the fewest requirements, in particular, the *uniqueness* metric is important because each chip must be uniquely identified!

Note that there is no security implications in this scenario, i.e., no attack model

Stability is not a critical concern as long as
1) bit flips are infrequent, i.e., intra-HD is relatively small, otherwise the probability of 'aliasing' gets unacceptably large
2) it is possible to use a 'fuzzy match' criteria when the identifier is read

• The PUF will be used to **authenticate** chips at periods throughout its lifetime

This usage scenario has more requirements because it is designed to protect against an attacker,

E.g., someone who wants to replace a chip in the field with a malicious clone

Practical Aspects of PUFs

- The PUF will be used to **authenticate** chips (cont.)

  So how does this work?

  At 'time 0' (after manufacturing), a trusted entity applies a process called **enrollment**

  Here, a subset of challenges are applied from the **much larger number of available challenges** and the challenge/response pairs (CRPs) are stored in a *secure database*

  Only the trusted entity has knowledge of which challenges are applied

  Later in the field, during the **authentication** process, the trusted entity removes CRPs from the database and applies them to the chip

  If the chip responds correctly with the same (or similar) response, we can assume it is the original chip, and not a malicious clone

Practical Aspects of PUFs

- The PUF will be used to **authenticate** chips (cont.)

Let's look more carefully at the assumptions

In this model, we assume that the PUF is able to produce a large set of CRPs

Otherwise, the number of CRPs in the secure database, and available to apply to the chip, will be small, significantly simplifing the task of creating a clone

We also assume that any CRPs that are used for authentication are NEVER used again (to avoid **replay attacks**)

Given that a *trusted party* is involved in authentication, the *stability* requirement can be relaxed

This is true b/c the trusted entity can apply 'fuzzy matching' criteria to validate the responses, i.e., the responses can be deemed 'close enough'

Bear in mind the intra-chip HD **must still be small** otherwise the 'code space' occupied by each chip becomes unacceptably large making it easier for the adversary to guess or clone it!

Practical Aspects of PUFs

- The PUF will be used to **authenticate** chips (cont.)

  However, *uniqueness* and *randomness* are still very important,

  I.e., it must be VERY difficult for an attacker to predict the response from a challenge and/or to build a clone

  Note that in order to implement this, it must be possible to apply challenges and receive responses from the chip (either in person or over the internet)

  I.e., the responses must be made available through I/O pins on the chip

  Also, there is no need to store any type of information on the chip, in, e.g., non-volatile memory (NVM)

- The PUF will be used to **generate encryption keys**

  The 'Creme de la creme' of the applications!

  Here, *uniqueness*, *randomness* and *stability* are all critical

Practical Aspects of PUFs

- The PUF will be used to **generate encryption keys**

How does this work?

The PUF needs to be able to, periodically, produce or re-produce the/a key, e.g., after power cycles of the chip

- If only one key is needed for the lifetime of the chip, then the 'challenges' can be hardcoded in an LFSR
- If a new key is needed periodically, then NVM is required to (at least) store the 'seed' used by the challenge generator on chip

There is NO tolerance for errors here -- even a difference of 1 bit in a 512-bit key causes total chaos between communicating parties

For public/private encryption engines, e.g., RSA and ECC, the private key need **not ever** be made public (can be 'hidden' on the chip)

Therefore, it is NOT necessary to have an I/O interface to the PUF!

Practical Aspects of PUFs

- The PUF will be used to **generate encryption keys**

  Note, there is no 'trusted entity' here, and the PUF engine on the chip must be able to run correctly without intervention

In conclusion, all applications require uniqueness and randomness:

- Chip ID: PUF bitstrings must be large enough to support the number of chips in the population. Intra-chip HD can be > 0 but bear in mind, number of unique IDs are reduced.
- Authentication: Requires large numbers of CRPs to prevent adversary from measuring them all and building a clone. Also subject to model building attacks b/c the responses are sent off-chip.
- Encryption: Large number of CRPs is not necessary in cases where only a single key (or small number of keys) need to be generated over lifetime of chip. Intra-chip HD must be zero (or a mechanism put in place to ensure it, as we discuss next).

Practical Aspects of PUFs

Perhaps the biggest challenge the encryption key scenario poses deals with 'stability'

The 'zero tolerance' criteria to **bit flips** makes it necessary to integrate additional mechanisms into the PUF engine

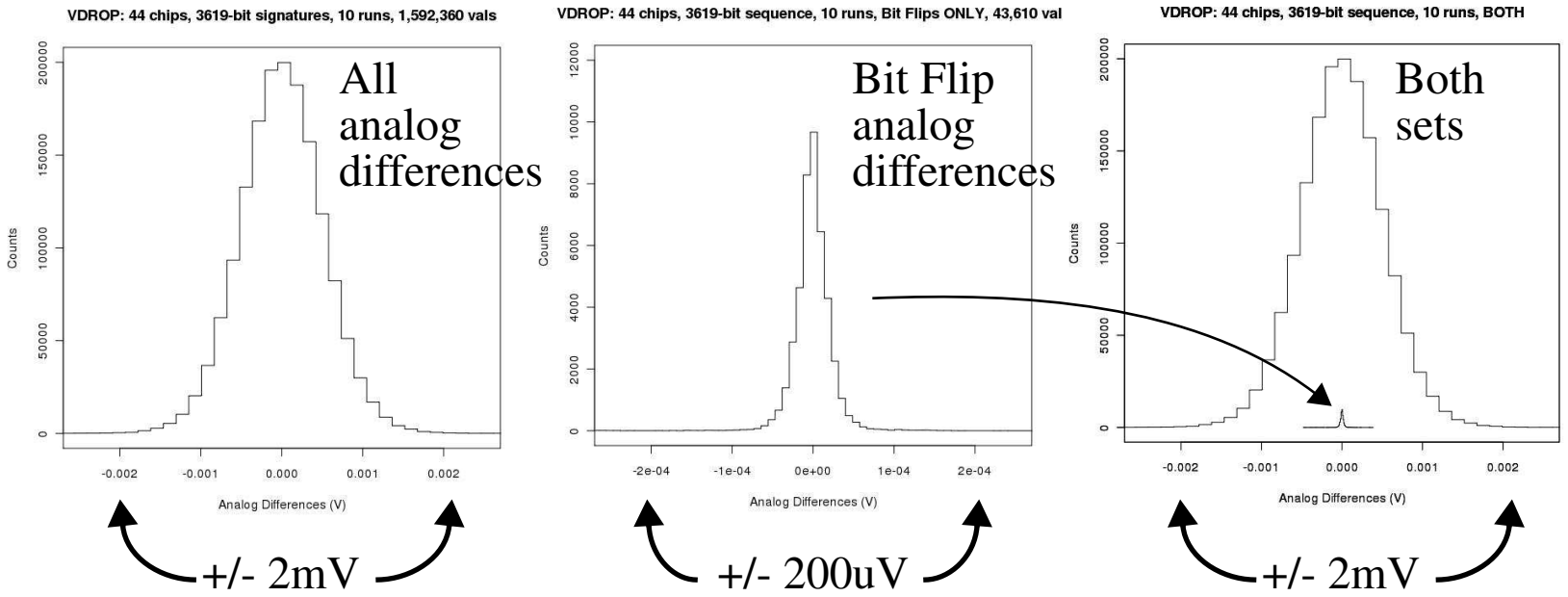
Bit flips occur when the two *analog quantities*, e.g., delays, leakage current, RO frequencies, voltages, are too similar to distinguish reliably in the presence of environmental 'noise' sources:

- Temperature variations
- Supply voltage variations
- EMI coupling

The nature of the 'unbiased' distributions (usually Guassian) of the analog values increases the probability that the two values being compared are VERY similar

Practical Aspects of PUFs

For example, consider histogram of the voltage differences (that when compared produce a digital ’0’ or ’1’ in the bit stream) from our power grid PUF



You might say, "why don’t you just design the PUF to ensure the analog quantities are always large enough to avoid this situation?"

In order to accomplish this, you would need to introduce ’BIAS’

This, in turn, would **destroy** *uniqueness*, i.e., all chips would produce similar, or worst case, the same, responses

Practical Aspects of PUFs

There is **no** PUF, to my knowledge, that can avoid the 'bit flip' problem, and it MUST be dealt with for crypto applications

Most deal with bit flips using **error correction**

Devadas et al. at MIT have quite a few publications which investigate schemes such as Indexed Based Syndrome (IBS) and BCH coding

Although they are designed to ensure error-free key regeneration, they come at a cost:

- Area overhead to implement the error correction algorithms
- Information leakage

The latter occurs because, in order to correct errors, you typically have to publically store "special" information that relates to the key

So in some sense, you are making it easier for an adversary to accomplish his/her goal!

Practical Aspects of PUFs

You might (eventually) deduce that perhaps you can setup a 'threshold' mechanism to deal with bit flips

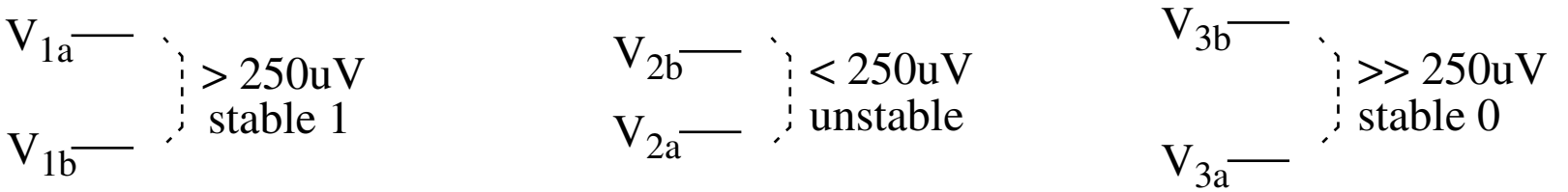
For example, from the analog voltage difference histograms shown earlier, you might be tempted to suggest the following:

"Everytime I re-generate the key, I'll compare the analog values for each challenge and discard (throw away) those that are less than my threshold"

For the sake of argument, let's assume you determine that 250 uV is a good threshold

Here's the caveat (we call it the *two-fence* problem):

Let's assume we apply a set of three challenges **at time 0** and measure the analog voltages as shown below



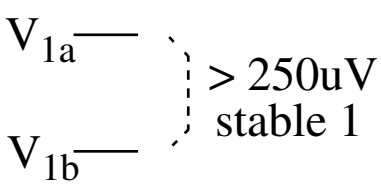
So you discard the 2nd bit b/c the analog difference is less than the threshold

Practical Aspects of PUFs

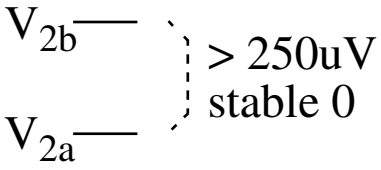
The fact that you only generated 2 of the 3 bits requires that you apply another challenge (not shown) -- let's say it produces a stable '1' and the final sequence is '101'

What can go wrong with this scheme?

Let's assume we need to re-generate the exact same key after a power cycle -- we apply the same challenges again and get:

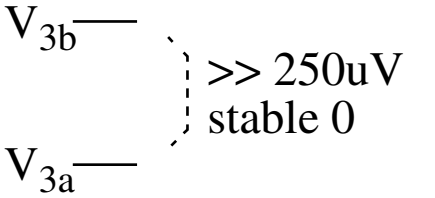


Challenge 1



Challenge 2

KEEP



Challenge 3

Ouch! Since the second challenge produced two analog voltages that are close to the threshold, it can happen that in some cases, it is *unstable* while in others it is *stable*

So, the final sequence becomes '100' this time -- NOT GOOD

One solution is to 'record' the results of the stability analysis during enrollment and store it somewhere (off-chip is fine) so the sequence can be replayed during regen.

Practical Aspects of PUFs

One last issue I want to bring up deals with *entropy*

All PUFs leverage physical sources of variation on the chip

- The primary difference among PUFs lies in the specific source they are each leveraging

For RO and arbiter PUFs, the primary source leveraged is variations in transistor threshold voltages

- They are **measured** *indirectly* by measuring the corresponding changes they produce in delay

Our power grid PUF leverages resistance variations, which we measure indirectly as voltage variations

Since the sources of physical variations are limited on the chip, most PUFs need to **re-use** these physical sources over-and-over again in the bit generation process

- For example, it is not uncommon to compare n sources of variation to produce $n*(n-1)/2$ bits

Practical Aspects of PUFs

Given this is true, the **goal** of an adversary, who is trying to predict the response of a PUF, is to *learn* the values of the constituent physical sources of variation

(Note: We assume authentication scenario where responses are sent off-chip, which doesn't happen for crypto apps).

Once these are known, e.g., once the frequencies of the individual ROs are 'learned', then it becomes possible to *predict the responses* to ANY challenge

It follows then that the **larger the source of physical variations**, the more difficult it is for an adversary to *build a model* of the PUF

SRAM, RO, VT and arbiter PUFs are limited to the 2-D space of the chip

On the other hand, a PUF derived from resistance variations in metal (and/or polysilicon) wires can leverage a **third** dimension and potentially be more resistant to *model building attacks*