Institute of Operating Systems
and Computer Networks

Technische
Universität
Braunschweig

Secure communication based on noisy input data

Physically unclonable functions

Stephan Sigg

June 28, 2011

Overview and Structure

Outline

Physical random functions

Physical random functions / Physically unclonable functions:
Random functions that can only be evaluated with the help of a physical system

Definition

A PUF is a random function that can only be evaluated with the help of a specific physical system. The inputs to a physical random function are challenges and the outputs are responses.

Physical random functions

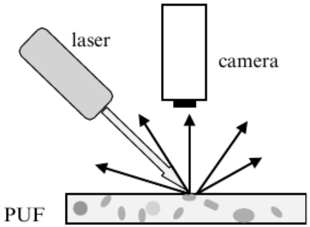
Digital PUFs Simplest kind of PUF. Digital key K is embedded in a tamper-proof package along with some logic that computes

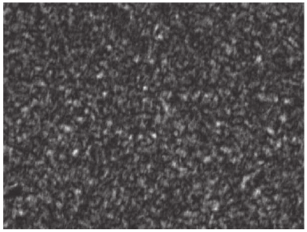
$$Response = RF(K, Challenge)$$

for some random function *RF*

Physical random functions

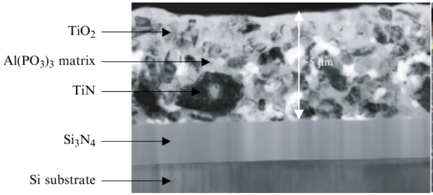
Optical PUFs Made of transparent optical medium containing bubbles. Shining a laser beam through the medium produces speckle pattern (response) that depends on exact position/direction of incoming beam.

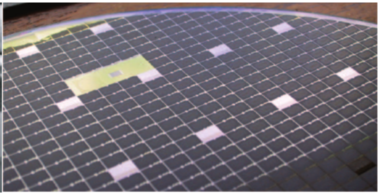




Physical random functions

Silicon PUFs Challenge is an input to a circuit that reconfigures the path that signals follow through the circuit. Response is related to the time it takes for signals to propagate through a complex circuit.





Physical random functions

Security of PUFs relies on difficulty of extracting all necessary parameters from a complex physical system

Attacker trying to extract all physical parameters might modify the PUF in the process

This makes PUFs tamper resistant to some extend

Physical random functions

PUF implementations build on random manufacturing variations (bubble position or exact wire delays):
Exact behaviour is a mystery even for the manufacturer

Not feasible to create two identical copies of a PUF

A difficulty of optical and silicon PUFs is that their output is noisy

Error correction that does not compromise the security is required[1]

[1] G.E. Suh, C.W. O'Donnell, I. Sachdev, S. Devadas, Design and implementation of the AEGIS single-chip secure processor using physical random functions, Proceedings of the 32nd Annual International Symposium of computer

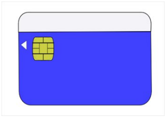
Physical random functions

Standard application: Key-card[2]

Lock stores a database of challenge response pairs (CRPs) for PUF

When the bearer of the PUF wants to open the lock, it selects a challenges it knows and asks the PUF for the corresponding response

Each CRP can be used only once : Card will eventually run out of PUFs



[2] R. P [illegible] Physical One-Way Functions, PhD thesis, MIT, 2001

Outline

Controlled Physical random functions

Definition

Controlled physical random function (CPUF):

PUF that can only be accessed through specific API

Main problem with uncontrolled PUFs: Anybody can query the PUF for the response to any challenge

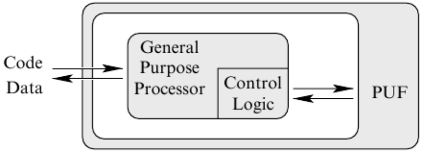
In order to engage in cryptography with a PUF device, a user has to exploit the fact that only he and the device know the response to a specific challenge.

Controlled Physical random functions

Third party could try to overhear challenge, obtain response from PUF and spoof the device

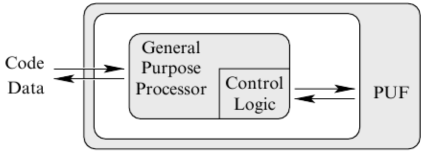
Problem: Adversary can freely query the PUF

By using CPUFs, Access to PUF restricted by control algorithm that prevents this attack



Embedding control logic for PUF in physical system of PUF makes it difficult to conduct invasive attacks on the control logic

Controlled Physical random functions



The PUF and its control logic have <u>complementary roles</u>

The PUF protects the control logic from invasive attacks

The control logic protects the PUF from protocol attacks

Controlled Physical random functions

Applications for CPUFs

Applications for CPUFs include applications that require single symmetric key on a chip

- Smartcards that implement authentication:

  Current smart-cards: Hidden digital keys can be extracted using various attacks

  PUF on the smartcard: Can authenticate chip – Digital key not required (Smartcard hardware itself is the secret key)

  Key can not be duplicated: Person that temporary looses control of card need not fear that an adversary might have cloned the card or that the security became somehow impaired.

Outline

CPUF API

CPUF typically modelled as general-purpose processing element with access to a PUF

Man-in-the-Middle Attack:
Adversary intercepts communication to device wants Alice to accept incorrect result as coming from device

Alice would execute the following protocol

CPUF API

Alice would execute the following protocol

1. Pick one CRP (`Char, Response`) at random
2. Execute the following function on the PUF:

```
GetAuthenticBroken(Chal){
        my Resp = PUF(Chal);
        // Do some computation, produce result
        return (Result, MAC(Result, Resp));
}
```

3. Use the MAC and Response to check that the data is authentic

CPUF API

Protocol is not secure against Man-in-the-Middle attacks

Attacker could

1. Intercept message send to `GetAuthenticBroken` and extract `Chal`
2. Execute on the PUF:
```
StealResponse(Chal){
        return (PUF(Chal));
}
```
3. Forward Alice the message `MAC(FakeResult, Response)`
4. Since the MAC was computed with the correct response, Alice accepts `FakeResult`

CPUF API

Problem: When Alice releases her challenge, Adversary can ask PUF for corresponding response and impersonate PUF

Problem persists as long as the PUF freely provides responses

CPUF API

GetSecret

To solve this problem:

PUF shall only be accessed via call

`GetSecret(Chal)=Hash(PHashReg, PUF(Chal))`

PUF reveals combination of response and executed program instead of response

Since the Hash is a one-way function: Response not recovered easily

CPUF API

GetSecret

We alter the call of Alice accordingly:

```
GetAuthenticBroken(Chal){
         hashblock()({// HB
                 // Do some computation, produce result
        });
        my Secret = GetSecret(Chal);
        return (Result, MAC(Result, Secret));
}
```

CPUF API

GetSecret

Alice can now compute `Secret` from `Response` by computing `Hash(PHash(HB),Response)` to check the MAC

An adversary has no way of obtaining `Secret`

CPUF API

GetCRP

However, the solution presented may be too restrictive for Alice also

With no CRP:
No way for Alice to obtain one in the first place:
Device never reveals response

Possible solution: Primitive called `GetCRP` that

1. Picks a random challenge
2. Computes the response
3. Returns the response to the caller

When space of challenges large enough:
Unlikely that attacker can compute CRPs identical to Alice's

CPUF API

GetResponse

Problem: Random number generators often vulnerable to attacks

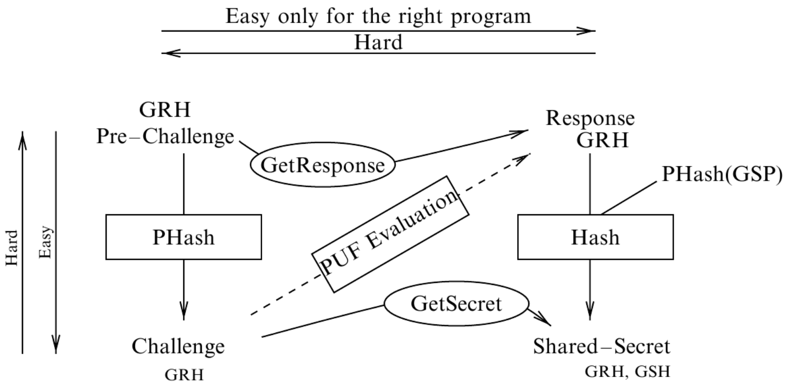
Therefore: Might prefer alternative that not relies on a RNG that much

Replace `GetCRP` by `GetResponse()=PUF(PHashReg)`

Now: Anybody can generate CRP `(PHashReg,GetResponse())`
But: Due to hash function, nobody can generate specific CRP

CPUF API

GetResponse



CPUF API

GetResponse

Man-in-the-Middle attack is prevented since each user has his own CRPs

Challenges can be public, but responses are required to be private

When not told the secret and GSH not leaks information, adversary can only obtain secret by hashing appropriate response

No way for adversary to obtain this response

Therefore:

Man-in-the-Middle attacks are prevented since PUF accessed only through `GetSecret` and `GetResponse`.

CPUF API

Challenge response pair management

How to get the response to the legitimate user?

The following sequence is proposed for CRP management

- After manufacturing manufacturer gets device-CRP with `Bootstrap`
- Manufacturer uses `Introduction` to provide CRPs to certification authorities
- Certification authorities provide CRPs to end users
- Anybody in possession of a CRP can create new CRPs by `Renew`

CPUF API

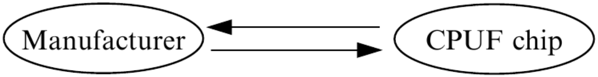
Bootstrapping

1. Pick a pre-challenge `PreChal` at random
2. Execute

```
Bootstrap(PreChal){
        hashblock(PreChal)({
                Return GetResponse();
        });
}
```

3. The challenge for the CRP is obtained by calculating `PHash(HB)`

If `PreChal` is not known, the security relies on the hash function



CPUF API

Renewal

1. Pick a pre-challenge `PreChal` at random
2. By using an old challenge `OldChal`, execute

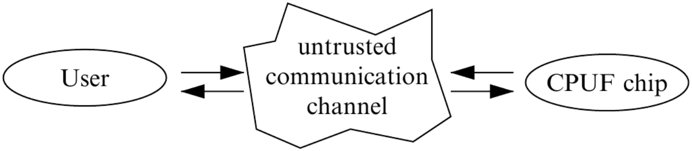
```
Renew(OldChal, PreChal){
        hashblock(OldChal, PreChal)({
                my NewResponse = GetResponse();
                my Secret = GetSecret(OldChal);
                return Encrypt(NewResponse, Secret);//Key:Secret
        });
}
```

3. Compute `Hash(PHash(HB), OldResponse)` to calculate `Secret`, check the MAC with it and retrieve `NewResponse`

CPUF API

Renewal

When the response corresponding to `OldChal` is only known to the user, the method is secure.



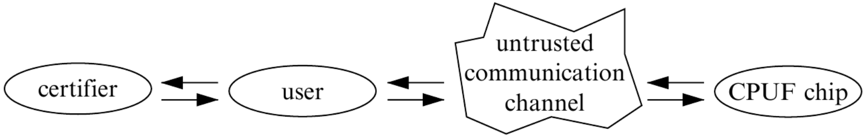
CPUF API

Introduction

Provide user with CRP

Assumption:

Trusted channel between user and certifier



CPUF API

Introduction

1. Cert. authority picks (OldChal, OldResponse), computes Secret=Hash(PHash(HB), OldResponse) and returns (OldChal, Secret)
2. User picks pre-challenge PreChal at random and executes

```
Introduction(OldChal, PubKey, PreChal){
        hashblock(PubKey, PreChal)({
                my NewResponse = GetResponse();
                my Message = PublicEncrypt(NewResponse, PubKey);
                my Secret' = GetSecret(OldChal);
                Return (Message, MAC(Message, Secret'));
        });
}
```

3. User checks MAC with Secret. (Secret=Secret' since both are computed as Hash(PHash(HB), OldResponse)). User Decrypts Message and computes PHash(HB) to obtain Response and Challenge

Outline

Questions?

Stephan Sigg
sigg@ibr.cs.tu-bs.de

Literature

C.M. Bishop: Pattern recognition and machine learning, Springer, 2007.

P. Tulys, B. Skoric, T. Kevenaar: Security with Noisy Data – On private biometrics, secure key storage and anti-counterfeiting, Springer, 2007.

W.W.Peterson, E.J. Weldon, Error-Correcting Codes, MIT press, 1972.

R.O. Duda, P.E. Hart, D.G. Stork: Pattern Classification, Wiley, 2001.