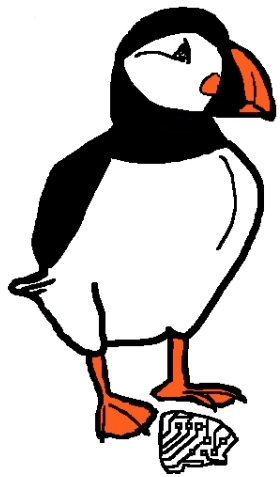
Physically unclonable functions found in standard PC components (PUFFIN)

Daniel J. Bernstein and Tanja Lange
reporting results of the PUFFIN project

http://puffin.eu.org
https://twitter.com/puffin_project



PUFFIN
INFSO-ICT-284833

2014.12.12

Physically Unclonable Functions

- Can uniquely identify electronic components.
- Device-unique "fingerprints" create root of trust in a hardware system; derive secret keys from physical properties.
- Protect valuable objects against counterfeiting.
- Typically found in specially designed hardware components.
- Fairly well understood are SRAM PUFs:
  - Microscopic manufacturing differences in SRAM determine whether a cell is more likely to hold a 0 or 1 when powered up.
  - Many cells are stable across reboots.
  - Need to be able to read up uninitialized memory.
  - Often found in specially designed hardware components, e.g., FPGA dongles.
- The same behavior generates true randomness.

PUFFIN Mission

Study and show the existence of SRAM PUFs and other types of PUFs in

- PCs,
- laptops,
- mobile phones,
- consumer electronics,

for use as secret keys or trust anchors in mass-market applications.
Show how to use this new root of trust to

- add security for mass-market applications,
- replace or complement the role of a trusted platform module,
- enable security for applications such as broadcast applications, content protection for the gaming industry,
- secure day-to-day transactions for everyone.

The results of the project will allow for the first time an a priori open platform, the most difficult element to secure in an information-technology system today, to inherit security properties from its own identity and its intrinsic physical properties.

Partners

Technische Universiteit Eindhoven, the Netherlands (Co-ordinator)

- Daniel J. Bernstein, Tanja Lange, Ruben Niederhagen, Boris Skoric

Intrinsic ID, Netherlands

- Pim Tuyls (scientific coordinator), Vincent van der Leest

Katholieke Universiteit Leuven, Belgium

- Bart Preneel, Anthony van Herrewege, Frederic Vercauteren

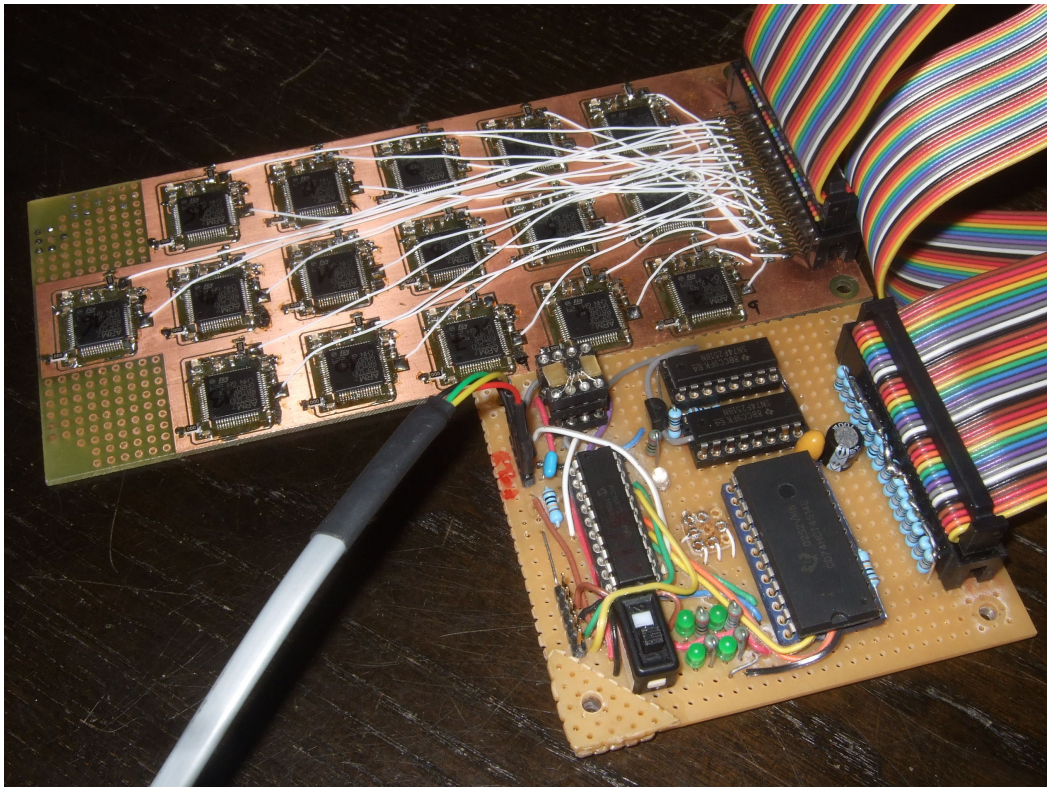
Technical University of Darmstadt, Germany

- Stefan Katzenbeisser, André Schaller

Scientific Work packages

- **WP1: Exploration**
  - Read out the uninitialized memory of various GPU and CPU types;
  - Make a preliminary assessment of the quality of the so obtained data;
  - Find identifying properties of mobile devices such as smart phones that are hard to clone;
  - Contingency plan: Consider PUFs on FPGAs as potential add-on.
- **WP2: Analysis and qualification**
  - Develop statistical analysis tools and mathematical and probabilistic models for the qualification of potential PUF data from WP1.
  - Run the tools: perform such analysis and qualification on WP1 data.
  - Recommend security parameters to use these PUFs in WP3.
- **WP3: Use cases**
  - Develop hardware-entangled cryptographic primitives that draw their security directly from physical assumptions of the underlying PUF.
  - Develop error correction schemes specifically tailored towards the error characteristics of the PUFs identified in WP1
  - Investigate to which extent the PUFFIN PUFs can be used to implement low-cost alternatives to Trusted Platform Modules.
  - Anti-counterfeiting: guarantee the integrity of software or securely bind software to a particular hardware platform.

STM32F100R8 microcontrollers (ARM Cortex-M3)

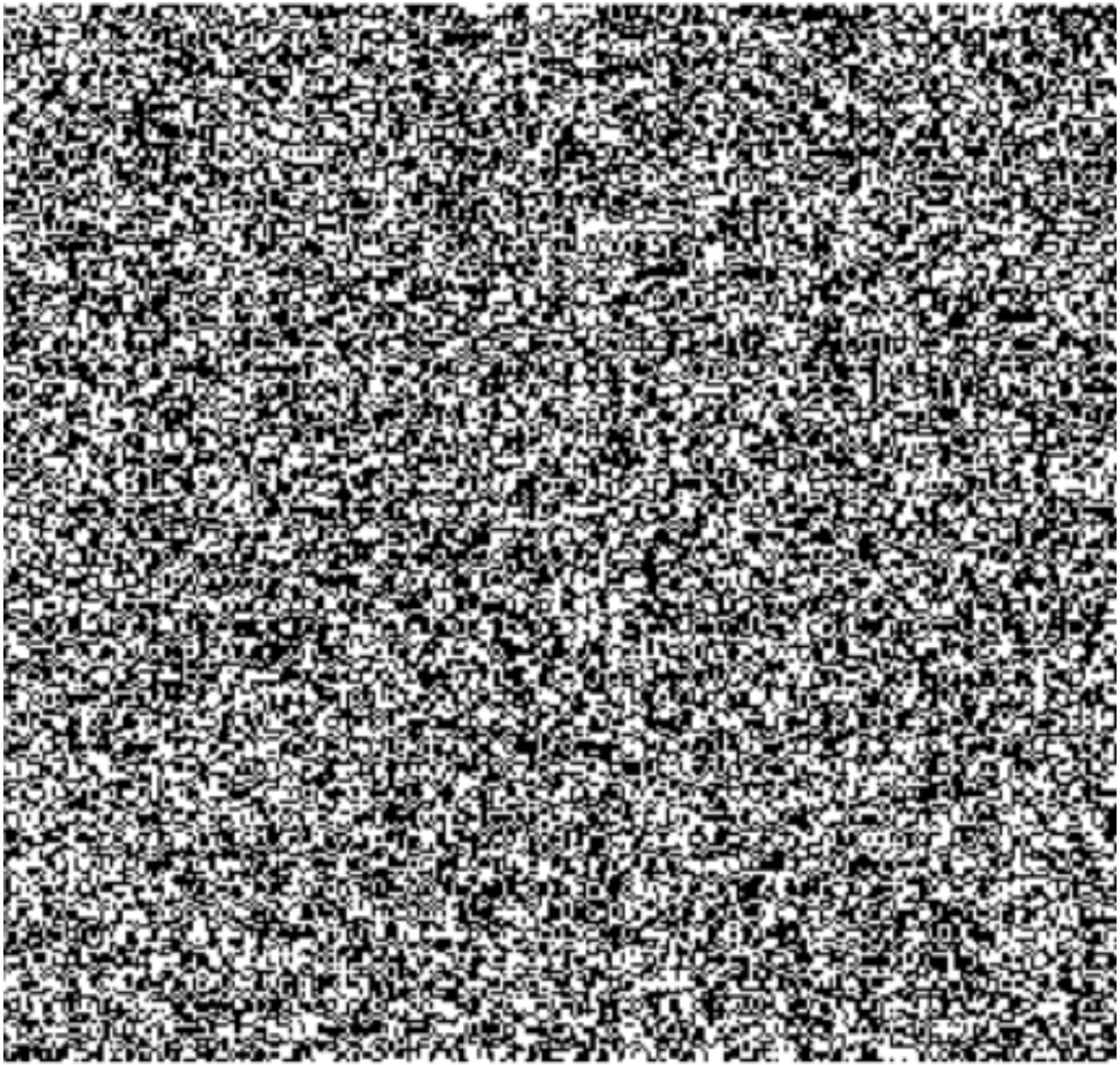


Custom PCB with several STM32F100R8 microcontrollers (32-bit ARM Cortex M3) and measurement board.

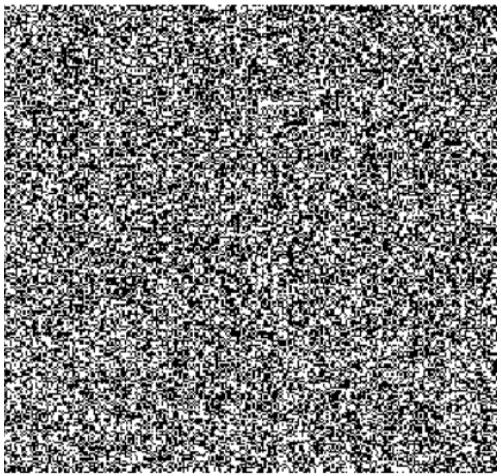
Programming pen

Simplify programming the microcontroller:
Six pogo pins contact the PCB; other end connects via USB to host PC.

STM32F100R8



Qualification of PUFs

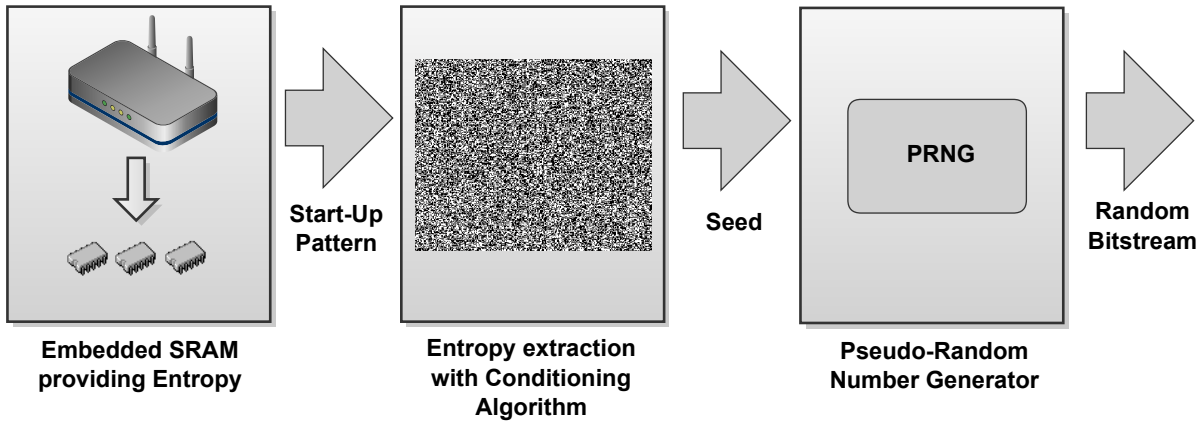


- Want enough stable bits to reconstruct secret.
- Stability is studied relative to one enrollment measurement.
- If all bits are stable this cannot be related to physical process but is programmed, so want some intermediate value.
- Require device identification, i.e., big differences between devices.
- To reconstruct enrollment secret use some *helper data*; usually this involves linear error correction codes.
- Make sure that helper data does not reveal information about the secret (e.g. bit 3 is 0).

Secure PRNG Seeding on COTS [1] devices

General idea:

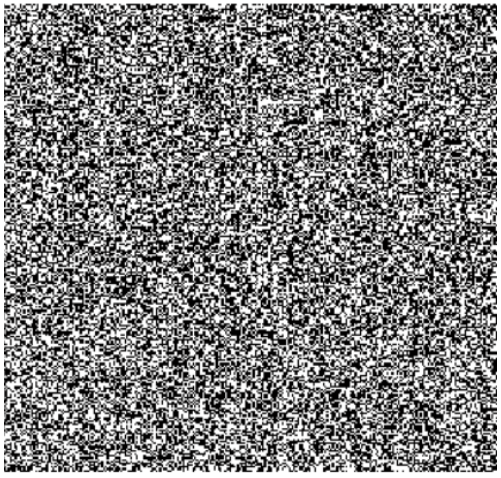
- Collect noisy SRAM bits upon early boot time.
- Overwrite SRAM bits to make it inaccessible during further usage.
- Apply hash function on noisy bits for entropy extraction.
- Use hash output as a seed for a PRNG.
- Use generated bit stream of random numbers for crypto applications.



[1] COTS: Commercial Off-The-Shelf

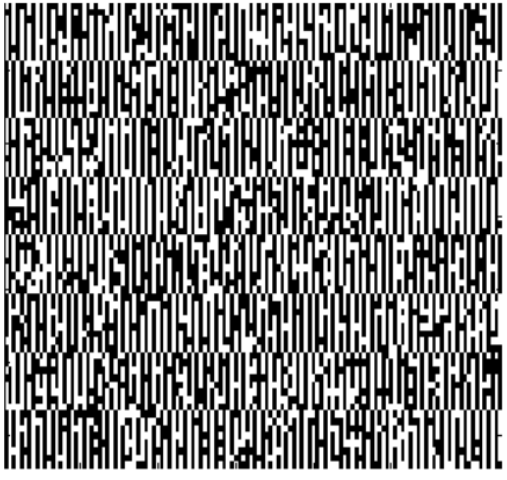
Secure PRNG Seeding on COTS Microcontrollers[2]

- STMicroelectronics STM32F100R8 with 8KiB SRAM.
- Contains enough entropy for TRNG: minimum 5,5%.
- We need about 1.04KiB to derive a truly random seed of 256 bits.



However . . .

- Microchip PIC16F1825 with 1 KiB SRAM.
- Startup values exhibit clearly visible *patterns*. Prediction attacks!
- Not enough entropy.



[2] Van Herrewege, van der Leest, Schaller, Katzenbeisser, Verbauwhede, TrustED'13

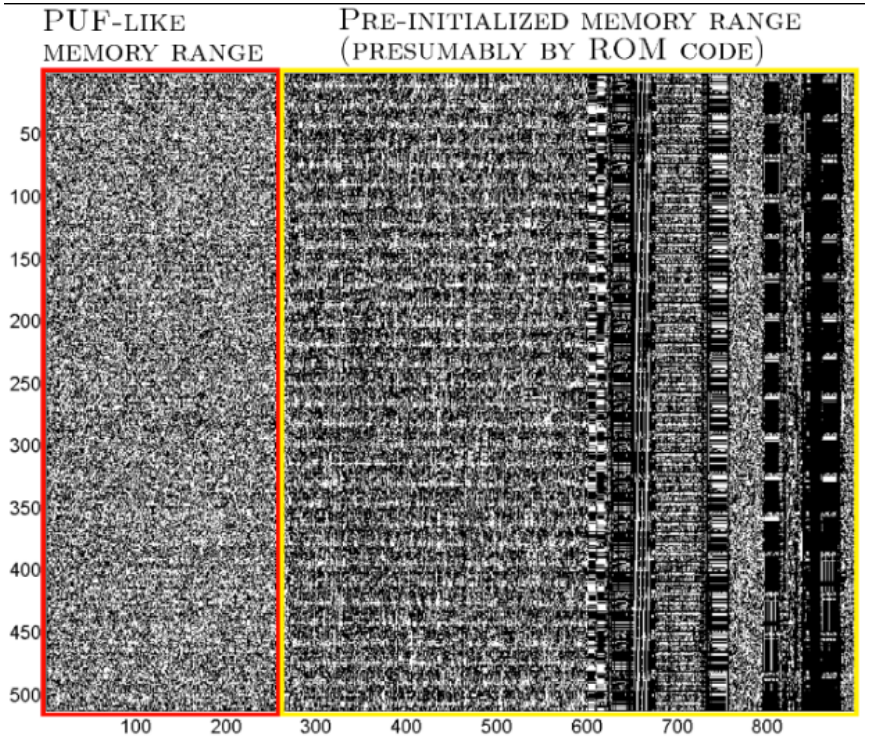
Light-Weight Secure Boot Implementation for SoCs

- Establish secure boot anchor on smart phones (link bootloader to device, stop malware at kernel level, might also enforce software license terms).
- PandaBoard has ARM Cortex-A9 (TI OMAP4460).

- PUF source: On-Chip Memory (OCM) L3 SRAM, 56 KiB
- OCM used to hold initializing code → part of the memory cannot be used as PUF as it is initialized.
- OCM part with good PUF behavior: ca. 12 KiB
- Enough to derive a 256 bit AES key.

Light-Weight Secure Boot for SoCs using PUFs[3]

- Hamming weight: ca. 49
- Max. within-class Hamming distance: 3.90 %
- Min. between-class Hamming distance: 50.02 %
- Current implementation needs 675 SRAM bytes
- Golay(23, 12, 7)-Code & repetition encoding for error correction



[3] Schaller, Arul, van der Leest

Secure Boot on Laptop or PC?

Typical boot sequence on AMD64 systems:

1. BIOS (UEFI),
2. boot loader,
3. operating system.

Secure Boot on Laptop or PC?

Typical boot sequence on AMD64 systems:

1. BIOS (UEFI),
2. boot loader,
3. operating system.

SRAM in AMD64 CPUs:

- General purpose registers.
- Vector registers (XMM): $16 \cdot 128 = 2048$ bits (per core).
- Caches.

Secure Boot on Laptop or PC?

Typical boot sequence on AMD64 systems:

1. BIOS (UEFI),
2. boot loader,
3. operating system.

SRAM in AMD64 CPUs:

- General purpose registers.
- Vector registers (XMM): $16 \cdot 128 = 2048$ bits (per core).
- Caches.

Registers on OS level: [small kernel patch and module]

- All XMM registers on CPU core 1 contained all 0.
- Some XMM registers on CPU core 0 contained, e.g.
  - `EFI_STATUS_CODE_SPECIFIC_DATA_GUID` and
  - `EFI_PROCESSOR_PRODUCER_GUID`.

Secure Boot on Laptop or PC?

Typical boot sequence on AMD64 systems:

1. BIOS (UEFI),
2. boot loader,
3. operating system.

SRAM in AMD64 CPUs:

- General purpose registers.
- Vector registers (XMM): $16 \cdot 128 = 2048$ bits (per core).
- Caches.

Registers on OS level: [small kernel patch and module]

- All XMM registers on CPU core 1 contained all 0.
- Some XMM registers on CPU core 0 contained, e.g.
  - `EFI_STATUS_CODE_SPECIFIC_DATA_GUID` and
  - `EFI_PROCESSOR_PRODUCER_GUID`.
- The registers have been initialized/used before the OS was started!

Registers on bootloader level

More serious code changes.

- All XMM registers except `xmm0` were 0.
- XMM register `xmm0` contained the same data on each boot, though different data for different test machines.
- The data turned out to be some fill-pattern of the BIOS code or some CPUID depending on the test machine.

Registers on bootloader level

More serious code changes.

- All XMM registers except `xmm0` were 0.
- XMM register `xmm0` contained the same data on each boot, though different data for different test machines.
- The data turned out to be some fill-pattern of the BIOS code or some CPUID depending on the test machine.
- The registers have been initialized/used before the OS was started!

Registers on bootloader level

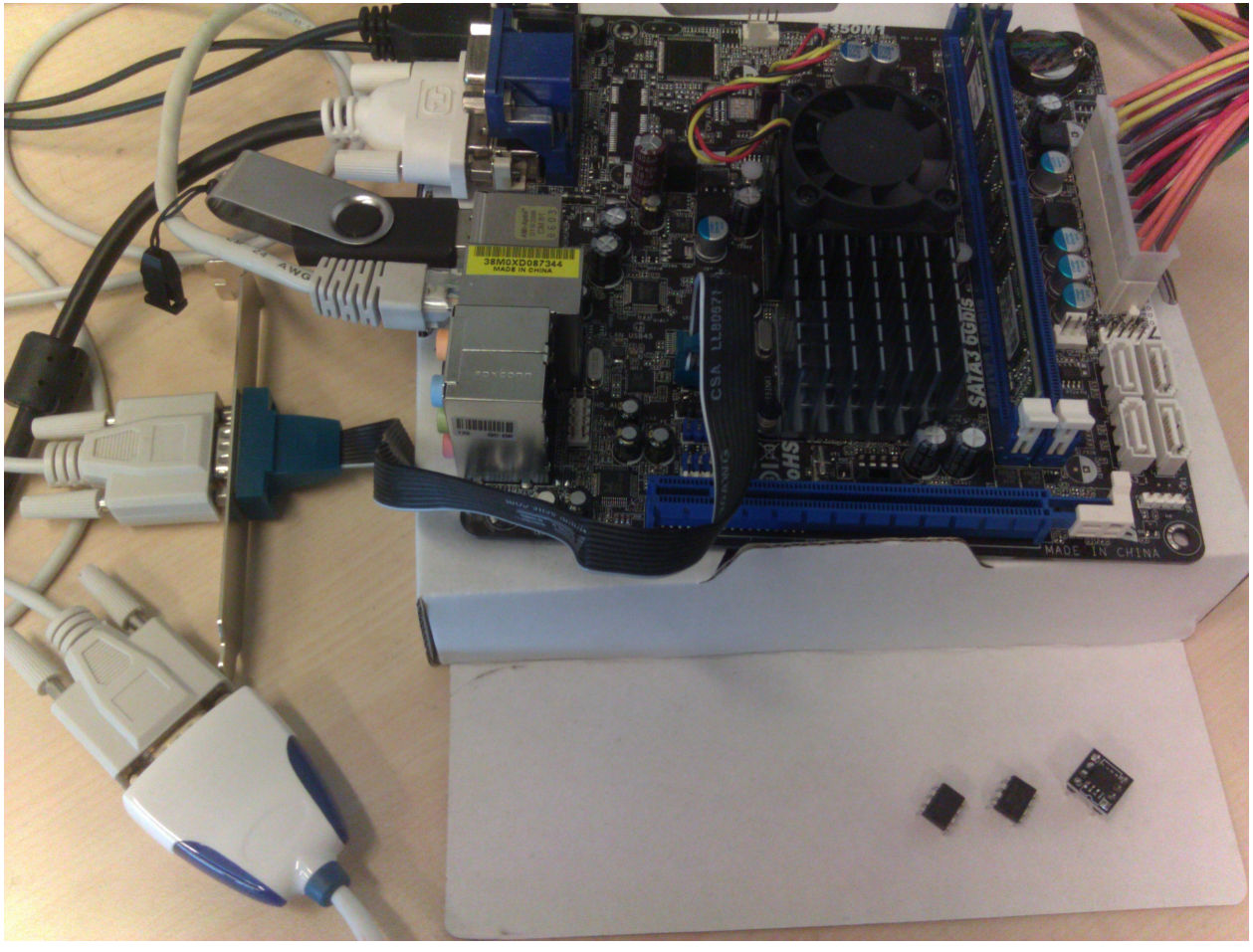
More serious code changes.

- All XMM registers except `xmm0` were 0.
- XMM register `xmm0` contained the same data on each boot, though different data for different test machines.
- The data turned out to be some fill-pattern of the BIOS code or some CPUID depending on the test machine.
- The registers have been initialized/used before the OS was started!

Try registers on BIOS level:

- Need to use Coreboot to access BIOS code.
- Only specific main boards are supported out-of-the-box.
- Read XMM registers as early as possible (before RAM is initialized).
- More serious Coreboot patch.
- Manual analysis of Coreboot disassembly ensures that (at least) `xmm2`–`xmm7` have not been touched before patch code.

ASRock E350M1 with AMD E-350 APU



Registers on BIOS Level

All XMM registers were 0.

Registers on BIOS Level

All XMM registers were 0.
Volume 1 of the *AMD64 Architecture Programmer's Manual* states:

> *Upon power-on reset, all 16 YMM/XMM registers are cleared to +0.0. However, initialization by means of the #INIT external input signal does not change the state of the YMM/XMM registers.*

Last chance:

Registers on BIOS Level

All XMM registers were 0.
Volume 1 of the *AMD64 Architecture Programmer's Manual* states:

> *Upon power-on reset, all 16 YMM/XMM registers are cleared to +0.0. However, initialization by means of the #INIT external input signal does not change the state of the YMM/XMM registers.*

Last chance:
Investigate the *cache* after power-on.

Cache on BIOS level

- Use Coreboot since we need access to BIOS code.
- Print cache-as-RAM region to serial console *before* RAM is initialized.
- Make sure cache-as-RAM is *not* nulled!
  $\Rightarrow$ Coreboot patch

Cache on BIOS level

- Use Coreboot since we need access to BIOS code.
- Print cache-as-RAM region to serial console *before* RAM is initialized.
- Make sure cache-as-RAM is *not* nulled!
  $\Rightarrow$ Coreboot patch
- Most data was 0,
- except for some data that had been stored to the stack previously.

Cache on BIOS level

- Use Coreboot since we need access to BIOS code.
- Print cache-as-RAM region to serial console *before* RAM is initialized.
- Make sure cache-as-RAM is *not* nulled!
  $\Rightarrow$ Coreboot patch
- Most data was 0,
- except for some data that had been stored to the stack previously.
- The cache has been initialized by hardware before BIOS code is executed!

Investigating SRAM PUFs in AMD64 CPUs[5]

- Bad news:

[4] van Aubel, Bernstein, and Niederhagen
[5] van Aubel and Niederhagen, submitted

Investigating SRAM PUFs in AMD64 CPUs[5]

- Bad news:
  On AMD64 CPUs, neither (XMM) registers nor caches allow access to uninitialized SRAM.

[4] van Aubel, Bernstein, and Niederhagen
[5] van Aubel and Niederhagen, submitted

Investigating SRAM PUFs in AMD64 CPUs[5]

- Bad news:
  On AMD64 CPUs, neither (XMM) registers nor caches allow access to uninitialized SRAM.
- More bad news:

[4] van Aubel, Bernstein, and Niederhagen
[5] van Aubel and Niederhagen, submitted

Investigating SRAM PUFs in AMD64 CPUs[5]

- Bad news:
  On AMD64 CPUs, neither (XMM) registers nor caches allow access to uninitialized SRAM.
- More bad news:
  It is not easy to publish a paper with negative results:

[4] van Aubel, Bernstein, and Niederhagen
[5] van Aubel and Niederhagen, submitted

Investigating SRAM PUFs in AMD64 CPUs[5]

- Bad news:
  On AMD64 CPUs, neither (XMM) registers nor caches allow access to uninitialized SRAM.
- More bad news:
  It is not easy to publish a paper with negative results:
- Good news:

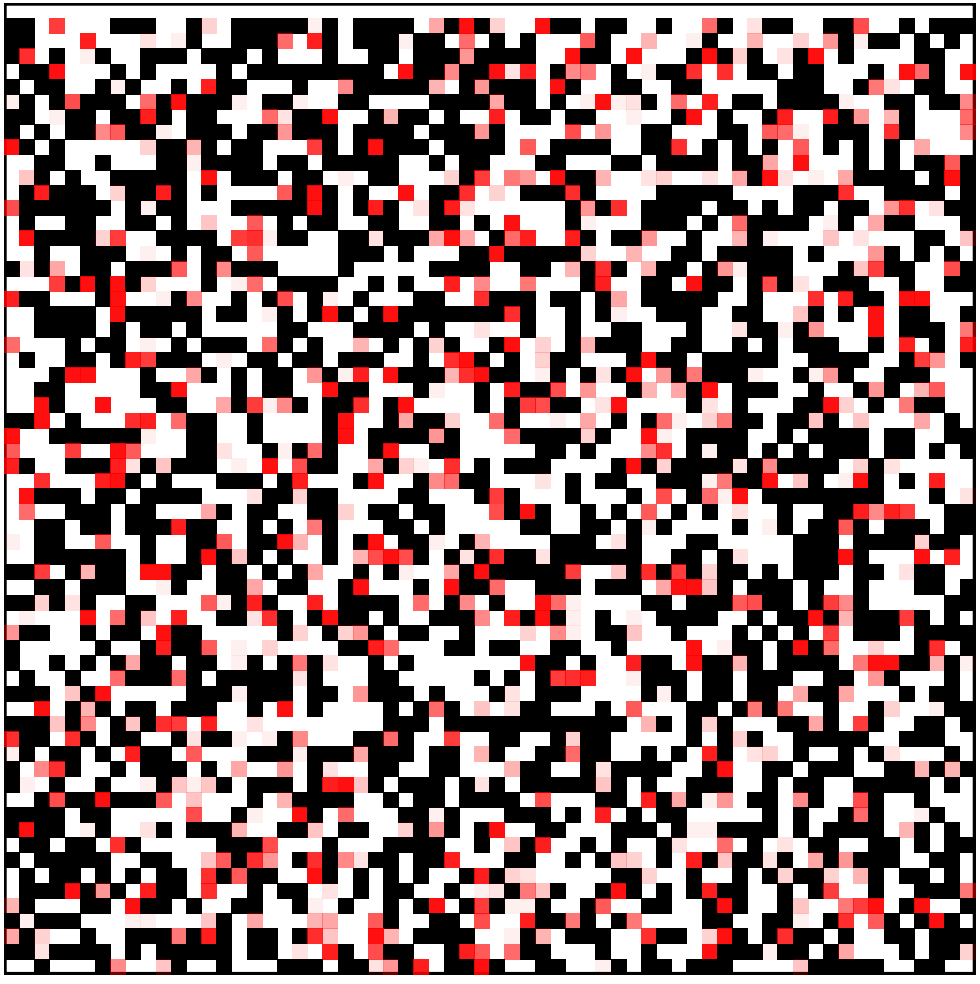
[4] van Aubel, Bernstein, and Niederhagen
[5] van Aubel and Niederhagen, submitted

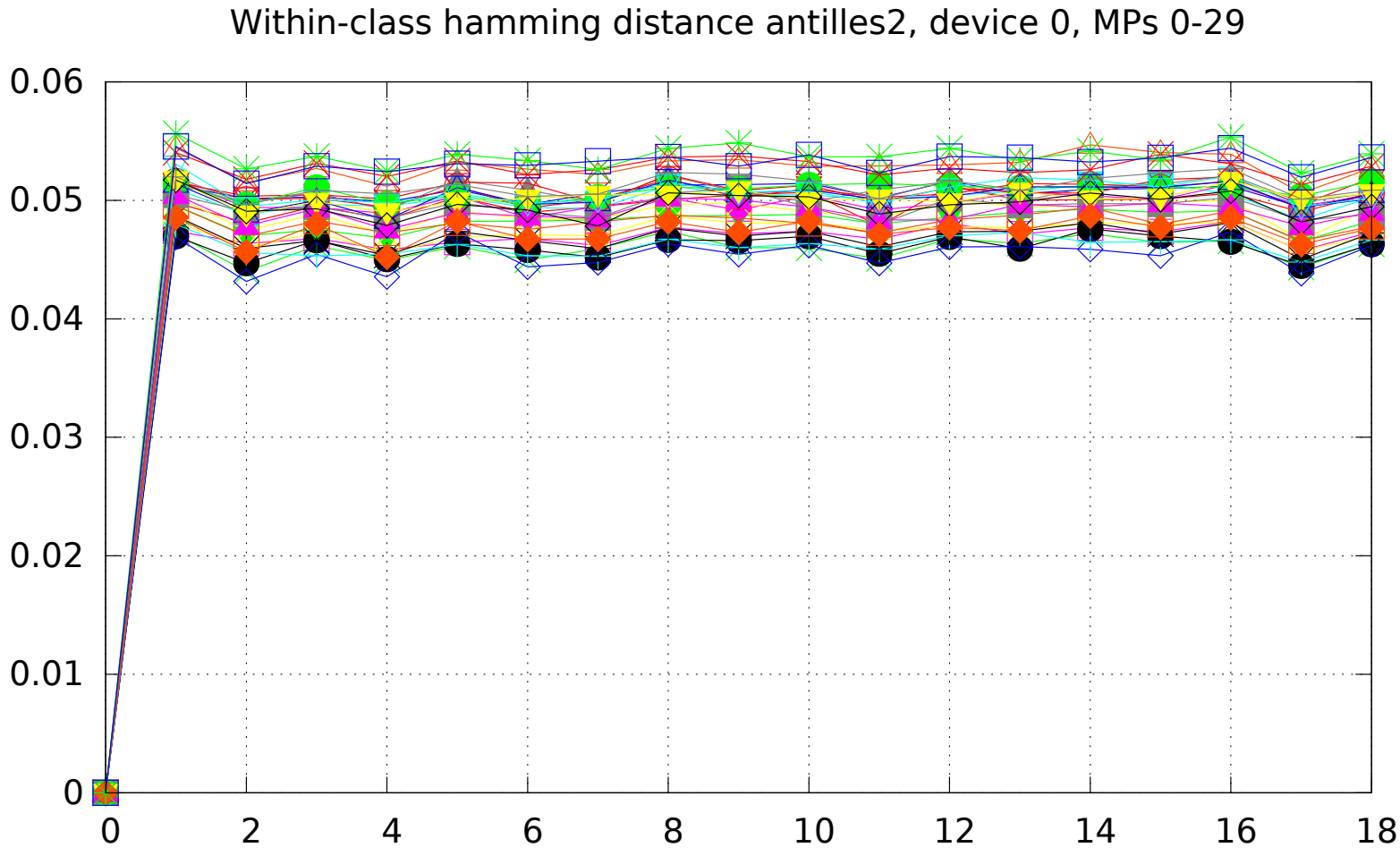
Investigating SRAM PUFs in AMD64 CPUs[5]

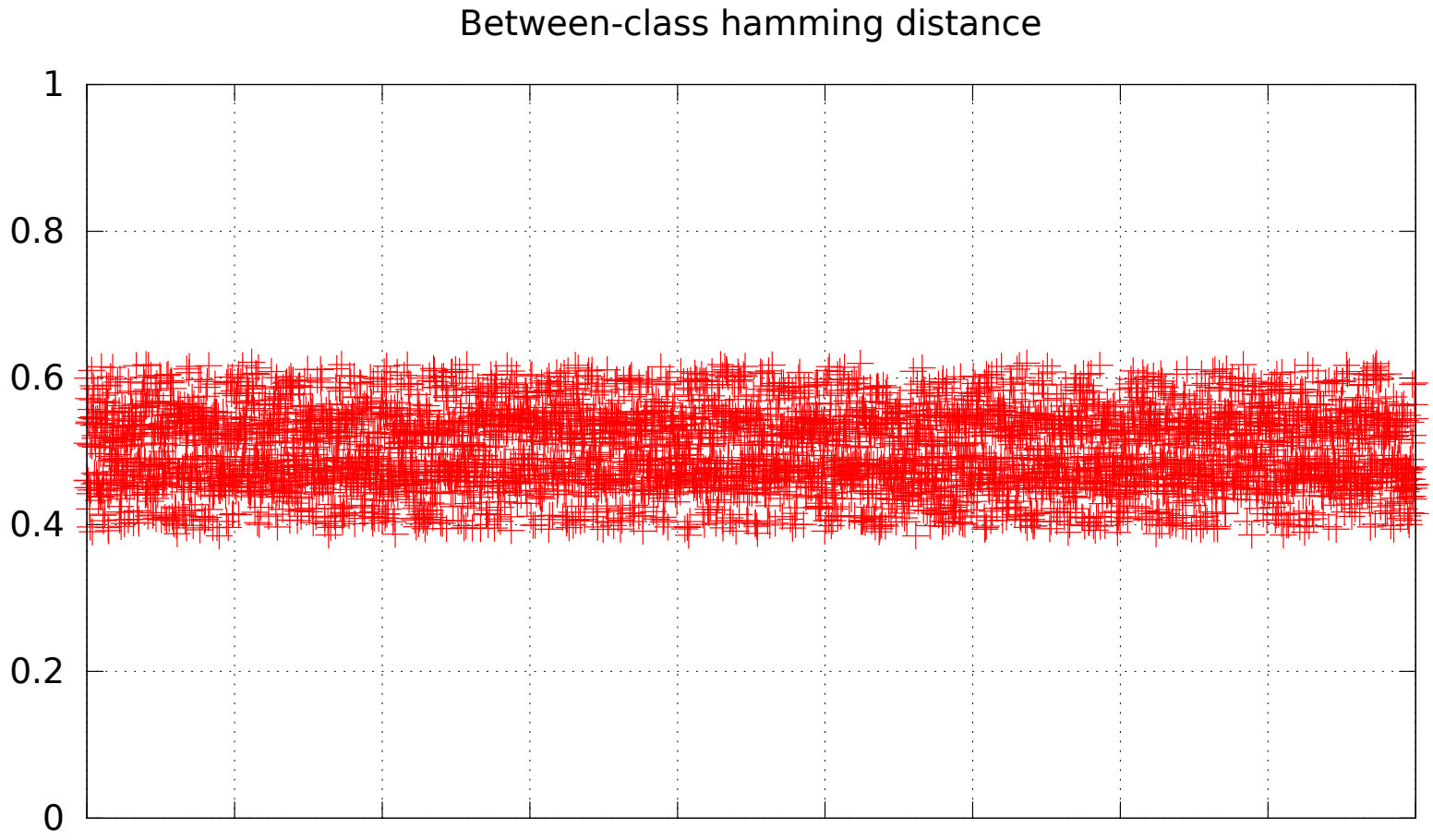
- Bad news:
  On AMD64 CPUs, neither (XMM) registers nor caches allow access to uninitialized SRAM.
- More bad news:
  It is not easy to publish a paper with negative results:
- Good news:
  NVIDIA GTX 295 GPUs exhibit PUF behavior.[4]
- NIVIDA GPUs are programmed using the CUDA framework.
- Experiments with 17 GTX 295 chips
  providing $17 \cdot 30 \cdot 16$ KB $= 8,160$ KB PUF data.

[4] van Aubel, Bernstein, and Niederhagen
[5] van Aubel and Niederhagen, submitted

antilles0, device 0, MP 0, 17 traces




Within-class hamming distance antilles2, device 0, MPs 0-29
0.06
0.05
0.04
0.03
0.02
0.01
0
0
2
4
6
8
10
12
14
16
18


Between-class hamming distance
1
0.8
0.6
0.4
0.2
0