Investigating SRAM PUFs in large CPUs and GPUs

Or: “Can’t we just rewrite the BIOS?”

SPACE 2015, MNIT, Jaipur, India

TU/e Technische Universiteit Eindhoven University of Technology

iCIS | Digital Security
Radboud University

Authors

Joint work:

Pol Van Aubel[1]
radboud@polvanaubel.com

[1] Radboud University
iCIS|Digital Security

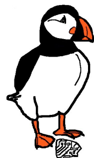
Daniel J. Bernstein[2,3]
djb@cr.yp.to

[2] University of Illinois at Chicago
Dept. of Computer Science

Ruben Niederhagen[3]
ruben@polycephaly.org

[3] Eindhoven University
of Technology

PUFFIN project



"The Physically unclonable functions found in standard PC components (PUFFIN) project intends to study and show the existence of SRAM PUFs and other types of PUFs in standard PCs, laptops, mobile phones and consumer electronics."
— http://puffin.eu.org/

Outline

Introduction to PUFs

CPUs

GPUs

Conclusions

Physically Unclonable Functions

PUFs:

Physically Unclonable Functions

PUFs:

- physical

Physically Unclonable Functions

PUFs:

- physical
- easy to evaluate

Physically Unclonable Functions

PUFs:

- physical
- easy to evaluate
- hard to characterize

Physically Unclonable Functions

PUFs:

- physical
- easy to evaluate
- hard to characterize
- easy to produce

Physically Unclonable Functions

PUFs:

- physical
- easy to evaluate
- hard to characterize
- easy to produce
- impossible to reproduce

Physically Unclonable Functions

PUFs:

- physical
- easy to evaluate
- hard to characterize
- easy to produce
- impossible to reproduce
- controlled (this message will self-destruct in 5 seconds)

Physically Unclonable Functions

PUFs:

- physical
- easy to evaluate
- hard to characterize
- easy to produce
- impossible to reproduce
- controlled (this message will self-destruct in 5 seconds)

Emergent behaviour through random physical variations.

Sources of PUFs

Inside ICs:

Sources of PUFs

Inside ICs:

- signal delay variations
  - ring oscillators
  - multiplexers

Sources of PUFs

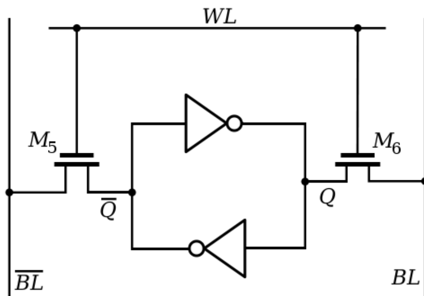
Inside ICs:

- signal delay variations
  - ring oscillators
  - multiplexers
- interaction between cross-coupled cells
  - SRAM
  - flip-flops
  - latches

Sources of PUFs

Inside ICs:

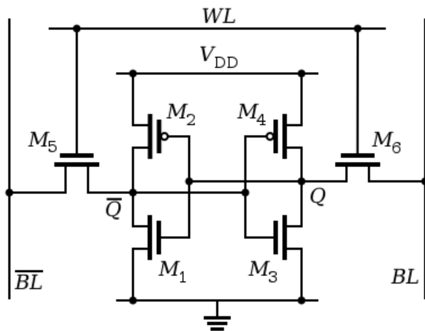
- signal delay variations
  - ring oscillators
  - multiplexers
- interaction between cross-coupled cells
  - SRAM
  - flip-flops
  - latches



Sources of PUFs

Inside ICs:

- signal delay variations
  - ring oscillators
  - multiplexers
- interaction between cross-coupled cells
  - SRAM
  - flip-flops
  - latches



Sources of PUFs

Outside ICs:

Sources of PUFs

Outside ICs:

- magnetism (magstripe)

Sources of PUFs

Outside ICs:

- magnetism (magstripe)
- metal resistance

Sources of PUFs

Outside ICs:

- magnetism (magstripe)
- metal resistance
- random capacitance coatings

Sources of PUFs

Outside ICs:

- magnetism (magstripe)
- metal resistance
- random capacitance coatings
- optical

Sources of PUFs

Outside ICs:

- magnetism (magstripe)
- metal resistance
- random capacitance coatings
- optical

Typically found in specially designed hardware components

SRAM PUFs

SRAM PUFs

- Microscopic differences determine likelihood of power-up state

SRAM PUFs

- Microscopic differences determine likelihood of power-up state
- Many cells are stable across reboots
  - Unique identification of electric components
  - Protect against counterfeiting
  - Device-unique “fingerprint” as a root of trust

SRAM PUFs

- Microscopic differences determine likelihood of power-up state
- Many cells are stable across reboots
  - Unique identification of electric components
  - Protect against counterfeiting
  - Device-unique “fingerprint” as a root of trust
- But not all; provides true randomness
  - Input for a CSRNG

SRAM PUFs

- Microscopic differences determine likelihood of power-up state
- Many cells are stable across reboots
  - Unique identification of electric components
  - Protect against counterfeiting
  - Device-unique "fingerprint" as a root of trust
- But not all; provides true randomness
  - Input for a CSRNG
- Already present in many devices as uninitialized memory

Outline

Introduction to PUFs

CPUs

GPUs

Conclusions

Targets

Modern, common, consumer-grade AMD64 CPUs:

Targets

Modern, common, consumer-grade AMD64 CPUs:

- AMD
- Intel

Targets

Modern, common, consumer-grade AMD64 CPUs:

- AMD
- Intel

High probability of SRAM used for registers and cache

Registers

AMD64 has many registers:

Registers

AMD64 has many registers:

- 16 64-bit General-Purpose (GP) registers

Registers

AMD64 has many registers:

- 16 64-bit General-Purpose (GP) registers
- 16 128-bit XMM-registers (used for SSE)

Registers

AMD64 has many registers:

- 16 64-bit General-Purpose (GP) registers
- 16 128-bit XMM-registers (used for SSE)
- Stuff like conditional registers, floating point / MMX ...

Registers

AMD64 has many registers:

- 16 64-bit General-Purpose (GP) registers
- 16 128-bit XMM-registers (used for SSE)
- Stuff like conditional registers, floating point / MMX ...

Easy to reach

Caches

Modern AMD64 CPUs have multiple layers of cache

Caches

Modern AMD64 CPUs have multiple layers of cache

- Transparent window into RAM
- AMD64 memory setup is complicated

Caches

Modern AMD64 CPUs have multiple layers of cache

- Transparent window into RAM
- AMD64 memory setup is complicated

Hard to reach

Security features

But CPUs evolved with security features:

- virtual memory
- address space separation
- memory protection

Security features

But CPUs evolved with security features:

- virtual memory
- address space separation
- memory protection

So can we actually read out uninitialized SRAM?

Security features

But CPUs evolved with security features:

- virtual memory
- address space separation
- memory protection

So can we actually read out uninitialized SRAM?

Well, maybe before the OS is really running?

AMD64 boot (somewhat simplified)

AMD64 boot (somewhat simplified)

1. Power on

AMD64 boot (somewhat simplified)

1. Power on
2. BIOS starts

AMD64 boot (somewhat simplified)

1. Power on
2. BIOS starts
3. ???

AMD64 boot (somewhat simplified)

1. Power on
2. BIOS starts
3. ???
4. Profit

AMD64 boot (somewhat simplified)

1. Power on
2. BIOS starts
3. ???
4. Profit

If we can avoid the ??? and move directly to profit:

AMD64 boot (somewhat simplified)

1. Power on
2. BIOS starts
3. ???
4. Profit

If we can avoid the ??? and move directly to profit:

- widely deployable

AMD64 boot (somewhat simplified)

1. Power on
2. BIOS starts
3. ???
4. Profit

If we can avoid the ??? and move directly to profit:

- widely deployable
- compatible with a lot of hardware

AMD64 boot (somewhat simplified)

1. Power on
2. BIOS starts
3. ???
4. Profit

If we can avoid the ??? and move directly to profit:

- widely deployable
- compatible with a lot of hardware
- least amount of effort

Uninitialized state access

Requirements:

Uninitialized state access

Requirements:

- Early

Uninitialized state access

Requirements:

- Early
- Readable

Uninitialized state access

Requirements:

- Early
- Readable
- Editable

Kernel patching

So you patch the Linux kernel:

Kernel patching

So you patch the Linux kernel:

- read and store XMM-registers as soon as they are available

Kernel patching

So you patch the Linux kernel:

- read and store XMM-registers as soon as they are available
- read that memory with a kernel module at a later point

Kernel patching

So you patch the Linux kernel:

- read and store XMM-registers as soon as they are available
- read that memory with a kernel module at a later point

Unfortunately, “that memory” was zeroes and e.g.

- EFI_STATUS_CODE_SPECIFIC_DATA_GUID
- EFI_PROCESSOR_PRODUCER_GUID

Kernel patching

So you patch the Linux kernel:

- read and store XMM-registers as soon as they are available
- read that memory with a kernel module at a later point

Unfortunately, "that memory" was zeroes and e.g.

- EFI_STATUS_CODE_SPECIFIC_DATA_GUID
- EFI_PROCESSOR_PRODUCER_GUID

So XMM-registers were modified before the kernel started

Power-on state access

Requirements:

- Earli***er***
- Readable
- Editable

Power-on state access

Requirements:

- Earli***er***
- Readable
- Editable

So can we actually read out uninitialized SRAM?

Power-on state access

Requirements:

- Earli***er***
- Readable
- Editable

So can we actually read out uninitialized SRAM?

Well, maybe early in the bootloader?

AMD64 boot (simplified)

1. Power on

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode
5. Run main BIOS / UEFI payload

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode
5. Run main BIOS / UEFI payload
6. Find bootable devices

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode
5. Run main BIOS / UEFI payload
6. Find bootable devices
7. Run bootsector code

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode
5. Run main BIOS / UEFI payload
6. Find bootable devices
7. Run bootsector code
8. More initialization and OS loading

AMD64 boot (simplified)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode
5. Run main BIOS / UEFI payload
6. Find bootable devices
7. Run bootsector code
8. More initialization and OS loading
9. Run OS kernel

GRUB

GRand Unified Bootloader

GRUB

GRand Unified Bootloader

- Installable on disk (easy edits)

GRUB

GRand Unified Bootloader

- Installable on disk (easy edits)
- Gets run immediately after boot-logic (BIOS or UEFI)

GRUB

GRand Unified Bootloader

- Installable on disk (easy edits)
- Gets run immediately after boot-logic (BIOS or UEFI)
- Runs in 32-bit protected mode: only 8 XMM-registers (still 1kb)

GRUB

GRand Unified Bootloader

- Installable on disk (easy edits)
- Gets run immediately after boot-logic (BIOS or UEFI)
- Runs in 32-bit protected mode: only 8 XMM-registers (still 1kb)
- Open source

Method

Take some random old Intel machine

Method

Take some random old Intel machine

1. Install linux

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:
   a. Figure out how GRUB works internally

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:
    a. Figure out how GRUB works internally
    b. Edit GRUB

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:
   a. Figure out how GRUB works internally
   b. Edit GRUB
   c. Compile GRUB

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:
    a. Figure out how GRUB works internally
    b. Edit GRUB
    c. Compile GRUB
    d. Install GRUB

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:
    a. Figure out how GRUB works internally
    b. Edit GRUB
    c. Compile GRUB
    d. Install GRUB
    e. Reboot

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:
    a. Figure out how GRUB works internally
    b. Edit GRUB
    c. Compile GRUB
    d. Install GRUB
    e. Reboot
    f. Pray

Method

Take some random old Intel machine

1. Install linux
2. Install vanilla grub
3. Clone GRUB
4. Until registers are read:
   a. Figure out how GRUB works internally
   b. Edit GRUB
   c. Compile GRUB
   d. Install GRUB
   e. Reboot
   f. Pray
   g. If praying fails: boot from install media and GOTO 2

GRUB internals

1. GRUB starts

GRUB internals

1. GRUB starts
2. Some machine initialization

GRUB internals

1. GRUB starts
2. Some machine initialization
3. Terminal initialization

GRUB internals

1. GRUB starts
2. Some machine initialization
3. Terminal initialization
4. Load modules

GRUB internals

1. GRUB starts
2. Some machine initialization
3. Terminal initialization
4. Load modules
5. Display boot menu

GRUB internals

1. GRUB starts
2. Some machine initialization
3. Terminal initialization
4. Load modules
5. Display boot menu
6. ...

GRUB internals

1. GRUB starts
2. Some machine initialization
3. Terminal initialization
4. Load modules
5. Display boot menu
6. ...

Doesn't seem to touch XMM-registers

GRUB internals, edited

1. GRUB starts
2. Some machine initialization
3. Terminal initialization
4. Read and dump XMM-registers to console
5. Load modules
6. Display boot menu
7. . . .

Reading registers from within GRUB

1. Allocate some memory

Reading registers from within GRUB

1. Allocate some memory
2. Fill with a known pattern

Reading registers from within GRUB

1. Allocate some memory
2. Fill with a known pattern
3. Use some ASM to copy each register

Reading registers from within GRUB

1. Allocate some memory
2. Fill with a known pattern
3. Use some ASM to copy each register
4. Dump memory to console

Reading registers from within GRUB

1. Allocate some memory
2. Fill with a known pattern
3. Use some ASM to copy each register
4. Dump memory to console
5. Sleep 60 seconds

Reading registers from within GRUB

1. Allocate some memory
2. Fill with a known pattern
3. Use some ASM to copy each register
4. Dump memory to console
5. Sleep 60 seconds while PhD-student writes furiously

Some gotchas

1. Allocate some memory
2. Fill with a known pattern
3. Explicitly set CPU in protected mode
4. Fix some other preconditions for SSE-instructions
5. Use some ASM to copy each register
6. Dump memory to console
7. Sleep 60 seconds while PhD-student writes furiously

Success! (Abort, retry, fail?)

XMM0: Some static data persisent over boots
XMM1–7: 0

Google

- XMM0 turns out to contain some CPUID-stuff
- Found in source files of the Coreboot project
- Likely explanation: BIOS uses XMM-registers

Power-on state access

Requirements:

- Earli***est***
- Readable
- Editable

Power-on state access

Requirements:

- Earli***est***
- Readable
- Editable

So can we actually read out uninitialized SRAM?

Power-on state access

Requirements:

- Earli***est***
- Readable
- Editable

So can we actually read out uninitialized SRAM?

Okay, okay, can't we just rewrite the BIOS?

AMD64 boot (simplified further)

1. Power on

AMD64 boot (simplified further)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM

AMD64 boot (simplified further)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware

AMD64 boot (simplified further)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode

AMD64 boot (simplified further)

1. Power on
2. Load BIOS / UEFI ROM from NVRAM
3. Initialize firmware
4. Put CPU in correct mode
5. Run main BIOS / UEFI payload

BIOS

BIOS is as early as you can get

BIOS

BIOS is as early as you can get
BIOS is not:

- Readable
- Editable

Coreboot

Formerly linuxBIOS

Coreboot

Formerly linuxBIOS

- Gets run first, contains boot-logic

Coreboot

Formerly linuxBIOS

- Gets run first, contains boot-logic
- Runs in 32-bit protected mode: much easier than 16-bit real mode

Coreboot

Formerly linuxBIOS

- Gets run first, contains boot-logic
- Runs in 32-bit protected mode: much easier than 16-bit real mode
- Open source

Coreboot

Formerly linuxBIOS

- Gets run first, contains boot-logic
- Runs in 32-bit protected mode: much easier than 16-bit real mode
- Open source

But:

- Each board booted by coreboot requires explicit support

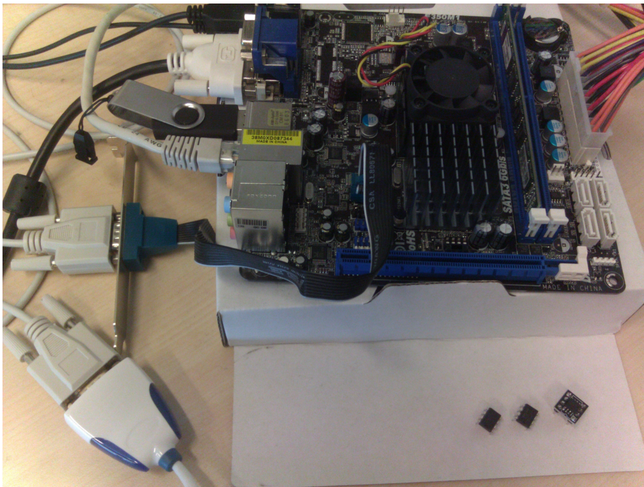
Not general

Port it ourselves or buy a supported board
Requirements:

1. Recent CPU
2. AMD or Intel
3. Relatively cheap
4. *Socketed* BIOS chip

ASROCK e350m1 w/ AMD e350 APU (CPU + Northbridge + stuff)
Some compatible BIOS chips

ASROCK e350m1



Method

Target registers first

1. Install linux
2. Clone coreboot
3. Get vanilla coreboot running on the board

Intermezzo: “compatible” chips aren’t

Clock speed “bug” in coreboot: BIOS chips incompatible
(I blame the chip vendor, not coreboot devs)
Last-minute trip to FOSDEM to exchange chips with coreboot devs

Method

Figure out how coreboot works

- Actually quite complex for something so limited in scope

AMD64 boot (simplified less)

1. Power on

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
7. Initialize Super-IO

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
7. Initialize Super-IO
8. Start outputting over serial port

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
7. Initialize Super-IO
8. Start outputting over serial port
9. ...

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
7. Store XMM-registers in memory
8. Initialize Super-IO
9. Start outputting over serial port
10. Dump XMM-registers to serial port
11. ...

Success! (Abort, retry, fail?)

XMM0–7: 0

But: manual analysis of coreboot disasm: xmm2-xmm7 are untouched before patch code path.

The importance of documentation

"Table 14-1 shows the initial processor state following either RESET or INIT. Except as indicated, processor resources generally are set to the same value after either RESET or INIT."

"SSE State XMM0–XMM15 = 0"

"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."

"Following a RESET (but not an INIT), all instruction and data caches are disabled, and their contents are invalidated (the MOESI state is set to the invalid state)."

Dead end?

They implement what they say in the documentation (unfortunately)

Dead end?

They implement what they say in the documentation (unfortunately)

Or do they?

Close examination of cache-as-ram initialization: explicit zeroing of allocated stack

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
    a. Fix preconditions for cache-as-ram
    b. Allocate stack
    c. Ensure stack is not zeroed

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
   a. Fix preconditions for cache-as-ram
   b. Allocate stack
   c. Ensure stack is not zeroed
7. Initialize Super-IO
8. Start outputting over serial port

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
    a. Fix preconditions for cache-as-ram
    b. Allocate stack
    c. Ensure stack is not zeroed
7. Initialize Super-IO
8. Start outputting over serial port
9. Dump entire stack to serial port

AMD64 boot (simplified less)

1. Power on
2. Load BIOS ROM from NVRAM
3. Jump to address 0xFFF0
4. Put CPU in protected mode
5. Do some CPU initialization
6. Initialize cache-as-ram for stack-based computing
   a. Fix preconditions for cache-as-ram
   b. Allocate stack
   c. Ensure stack is not zeroed
7. Initialize Super-IO
8. Start outputting over serial port
9. Dump entire stack to serial port
10. ...

Fail

Everything except some space used for function calls is 0

Fail

Everything except some space used for function calls is 0

So can we actually read out uninitialized SRAM?

Fail

Everything except some space used for function calls is 0

So can we actually read out uninitialized SRAM?

No.

Outline

Introduction to PUFs

CPUs

GPUs

Conclusions

Security features?

No protection like the CPUs

Targets

Modern, common, consumer-grade Nvidia GPUs: Nvidia GTX 295

- Two GPU devices per card, with
- 30 multiprocessors per device, with
- 16384 32-bit registers, and
- 16KiB shared memory

Targets

Modern, common, consumer-grade Nvidia GPUs: Nvidia GTX 295

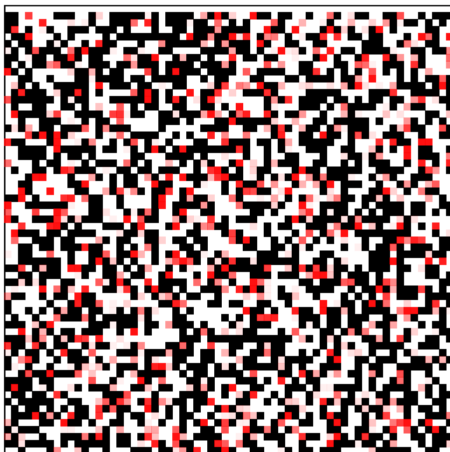
- Two GPU devices per card, with
- 30 multiprocessors per device, with
- 16384 32-bit registers, and
- 16KiB shared memory

High probability of SRAM used for registers and shared memory

Shared memory

- Easier to access than the registers
- We were able to read 490KiB of shared memory in each GPU, and repeated that on 17 devices
- Nice PUF properties
- No obstacles to building PUFs on these devices

Measurements / probabilities



Outline

Introduction to PUFs

CPUs

GPUs

Conclusions

Difficulties

- Decompiling and analyzing flow of BIOS code
- Compilers using XMM-registers as scratch-registers
- Ensuring negative results are not caused by human error
- Complexity of bringing up an AMD64 machine

Code

Available at
https://www.polvanaubel.com/research/puf/x86-64/code/[1]

[1] Actually, http://www.polvanaubel.com/research/puf/x86-64/code/ until I get a chance to fix it.