18:09 Memory Scrambling on Intel Sandy Bridge DDR3

by Nico Heijningen

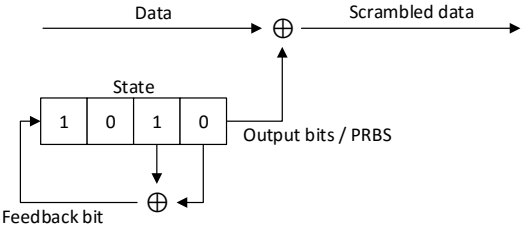
Humble greetings neighbors,

I reverse engineered part of the memory scrambling included in Intel's Sandy/Ivy Bridge processors. I have distilled my research in a PoC that can reproduce all 2^{18} possible 1,024 byte scrambler sequences from a 1,026 bit starting state.[50]

For a while now Intel's memory controllers include memory scrambling functionality. Intel's documentation explains the benefits of scrambling the data before it is written to memory for reducing power spikes and parasitic coupling.[51] Prior research on the topic[52] [53] quotes different Intel patents.[54]

Furthermore, some details can be deduced by cross-referencing datasheets of other architectures[55], for example the scrambler is initialized with a random 18 bit seed on every boot; the SCRMSEED. Other than this nothing is publicly known or documented by Intel. The prior work shows that scrambled memory can be descrambled, yet newer versions of the scrambler seem to raise the bar, together with prospects of full memory encryption.[56] While the scrambler has never been claimed to provide any cryptographic security, it is still nice to know how the scrambling mechanism works.

Not much is known as to the internals of the memory scrambler, Intel's patents discuss the use of LFSRs and the work of Bauer et al. has modeled the scrambler as a stream cipher with a short period. Hence the possibility of a plaintext attack to recover scrambled data: if you know part of the memory content you can obtain the cipher stream by XORing the scrambled memory with the plaintext. Once you know the cipher stream you can repetitively XOR this with the scrambled data to obtain the original unscrambled data.



An analysis of the properties of the cipher stream has to our knowledge never been performed. Here I will describe my journey in obtaining the cipher stream and analyzing it.

First we set out to reproduce the work of Bauer et al.: by performing a cold-boot attack we were able to obtain a copy of memory. However, because this is quite a tedious procedure, it is troublesome to profile different scrambler settings. Bauer's work is built on 'differential' scrambler images: scrambled with one SCRMSEED and descrambled with another. The data obtained by using the procedure of Bauer et al. contains some artifacts because of this.

We found that it is possible to disable the memory scrambler using an undocumented Intel register and used coreboot to set it early in the boot process. We patched coreboot to try and automate the process of profiling the scrambler. We chose the Sandy Bride platform as both Bauer et al.'s work was based on it and because coreboot's memory initialization code has been reverse engineered for the platform.[57] Although coreboot builds out-of-the-box for the Gigabyte GA-B75M-D3V motherboard we used, coreboot's makefile ecosystem is quite something to wrap your head around. The code contains some lines dedicated to the memory scrambler, setting the scrambling seed or SCRMSEED. I patched the code in Figure 28 to disable the

[50] `unzip pocorgtfo18.pdf IntelMemoryScrambler.zip`

[51] See for example Intel's 3rd generation processor family datasheet section 2.1.6 Data Scrambling.

[52] Johannes Bauer, Michael Gruhn, and Felix C. Freiling. "Lest we forget: Cold-boot attacks on scrambled DDR3 memory." In: Digital Investigation 16 (2016), S65–S74.

[53] Yitbarek, Salessawi Ferede, et al. "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors." High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on. IEEE, 2017.

[54] USA Patents 7945050, 8503678, and 9792246.

[55] See 24.1.45 DSCRMSEED of N-series Intel® Pentium® Processors and Intel® Celeron® Processors Datasheet – Volume 2 of 3, February 2016

[56] Both Intel and AMD have introduced their flavor of memory encryption.

[57] For most platforms the memory initialization code is only available as an blob from Intel.

```
3784 static void set_scrambling_seed(ramctr_timing * ctrl)
     {
3786   int channel;

3788   /* FIXME: we hardcode seeds. Do we need to use some PRNG for them?
          I don't think so. */
3790   static u32 seeds[NUM_CHANNELS][3] = {
         {0x00009a36, 0xbafcfdcf, 0x46d1ab68},
3792     {0x00028bfa, 0x53fe4b49, 0x19ed5483}
       };
3794   FOR_ALL_POPULATED_CHANNELS {
         MCHBAR32(0x4020 + 0x400 * channel) &= ~0x10000000;
3796     write32(DEFAULT_MCHBAR + 0x4034, seeds[channel][0]);
         write32(DEFAULT_MCHBAR + 0x403c, seeds[channel][1]);
3798     write32(DEFAULT_MCHBAR + 0x4038, seeds[channel][2]);
       }
3800 }
```

Figure 28. Coreboot's Scrambling Seed for Sandy Bridge

memory scrambler, write all zeroes to memory, reset the machine, enable the memory scrambler with a specific SCRMSEED, and print a specific memory region to the debug console. (COM port.) This way we are able to obtain the cipher stream for different SCRMSEEDs. For example when writing eight bytes of zeroes to the memory address starting at `0x10000070` with the scrambler disabled, we read `3A E0 9D 70 4E B8 27 5C` back from the same address once the PC is reset and the scrambler is enabled. We know that that's the cipher stream for that memory region. A reset is required as the SCRMSEED can no longer be changed nor the scrambler disabled after memory initialization has finished. (Registers need to be locked before the memory can be initialized.)

Now some leads by Bauer et al. based on the Intel patents quickly led us in the direction of analyzing the cipher stream as if it were the output of an LFSR. However, taking a look at any one of the cipher stream reveals a rather distinctive usage of a LFSR. It seems as if the complete internal state of the LFSR is used as the cipher stream for three shifts, after which the internal state is reset into a fresh starting state and shifted three times again. (See Figure 29.)

```
00111010 11100000
10011101 01110000
01001110 10111000
00100111 01011100
```

It is interesting to note that a feedback bit is being shifted in on every clocktick. Typically only the bit being shifted out of the LFSR would be used as part of the 'random' cipher stream being generated, instead of the LFSR's complete internal state. The latter no longer produces a random stream of data, the consequences of this are not known but it is probably done for performance optimization.

These properties could suggest multiple constructions. For example, layered LFSRs where one LFSR generates the next LFSR's starting state, and part of the latter's internal state being used as output. However, the actual construction is unknown. The number of combined LFSRs is not known, neither is their polynomial (positions of the feedback taps), nor their length, nor the manner in which they're combined.

Normally it would be possible to deduce such information by choosing a typical length, e.g. 16-bit, LFSR and applying the Berlekamp Massey algorithm. The algorithm uses the first 16-bits in the cipher stream and deduces which polynomials could possibly produce the next bits in the cipher stream. However, because of the previously described unknowns this leads us to a dead end. Back to the drawing board!

Automating the cipher stream acquisition by also patching coreboot to parse input from the serial console we were able to dynamically set the SCRMSEED, then obtain the cipher stream. Writing a Python script to control the PC via a serial cable enabled us to iterate all 2^{18} possible SCRMSEEDs and

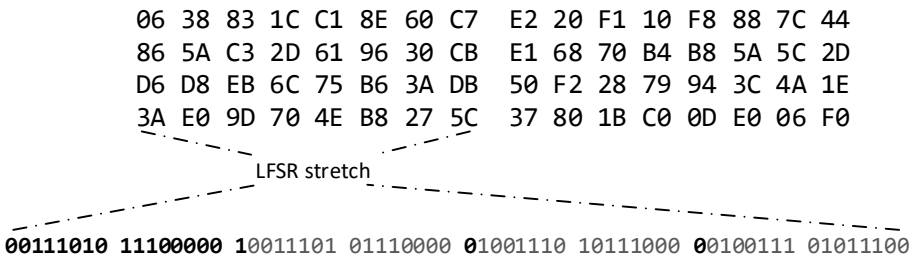


Figure 29. Keyblock

save their accompanying 1024 byte cipher streams. Acquiring all cipher streams took almost a full week. This data now allowed us to try and find relations between the SCRMSEED and the produced cipher stream. Stated differently, is it possible to reproduce the scrambler's working by using less than $2^{18} \times 1024$ bytes?

This analysis was eased once we stumbled upon a patent describing the use of the memory bus as a high speed interconnect, under the name of TeraDIMM.[58] Using the memory bus as such, one would only receive scrambled data on the other end, hence the data needs to be descrambled. The authors give away some of their knowledge on the subject: the cipher stream can be built from XORing specific regions of the stream together. This insight paved the way for our research into the memory scrambling.

The main distinction that the TeraDIMM patent makes is the scrambling applied is based on four bits of the memory address versus the scrambling based on the (18-bit) SCRMSEED. Both the memory address- and SCRMSEED-based scrambling are used to generate the cipher stream 64 byte blocks at a time.[59] Each 64 byte cipher-stream-block is a (linear) combination of different blocks of data that are selected with respect to the bits of the memory address. See Figure 30.

Because the address-based scrambling does not depend on the SCRMSEED, this is canceled out in the differential images obtained by Bauer. This is how far the TeraDIMM patent takes us; however, with this and our data in mind it was easy to see that the SCRMSEED based scrambling is also built up by XORing blocks together. Again depending on the bits of the SCRMSEED set, different blocks are XORed together.

Hence, to reproduce any possible cipher stream we only need four such blocks for the address scrambling, and eighteen blocks for the SCRMSEED scrambling. We have named the eighteen SCRMSEEDs that produce the latter blocks the (SCRMSEED) toggleseeds. We'll leave the four address scrambling blocks for now and focus on the toggleseeds.

The next step in distilling the redundancy in the cipher stream is to exploit the observation that for specific toggleseeds parts of the 64 byte blocks overlap in a sequential manner. (See Figure 32.) The 18 toggleseeds can be placed in four groups and any block of data associated with the toggleseeds can be reproduced by picking a different offset in the non-redundant stream of one of the four groups. Going back from the overlapping stream to the cipher stream of SCRMSEED `0x100` we start at an offset of 16 bytes and take 64 bytes, obtaining `00 30 80 ... 87 b7 c3`.

[58] US Patent 8713379.

[59] This is the largest amount of data that can be burst over the DDR3 bus.

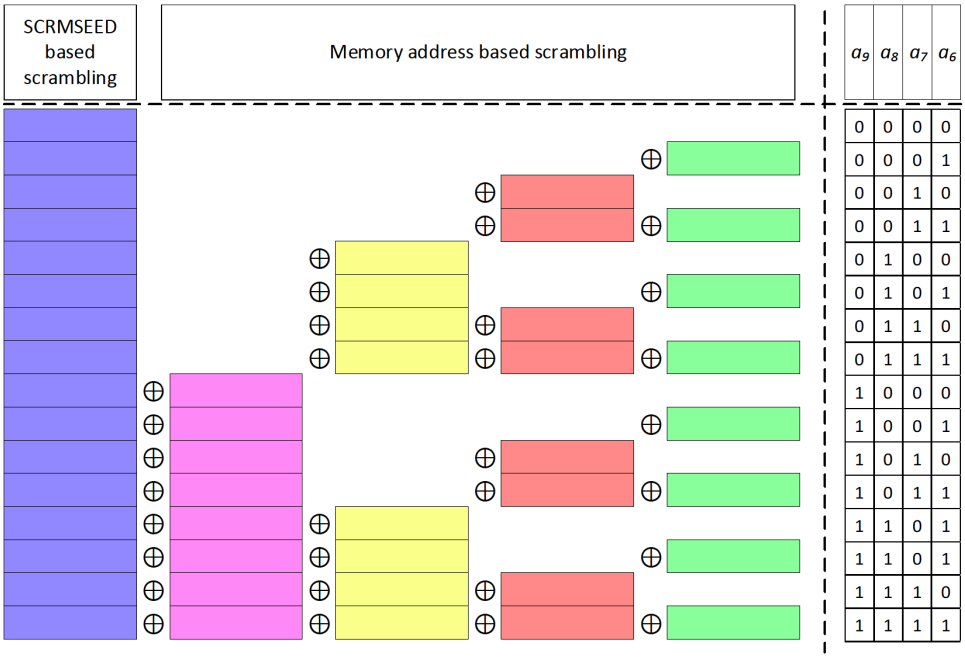


Figure 30. TeraDIMM Scrambling

$$\texttt{overlappingstream}(\textcircled{z}) \quad \begin{pmatrix} 0000\ 1100\ 0000 \\ 0000\ 0110\ 0000 \\ 0000\ 0011\ 0000 \\ 0000\ 0001\ 1000 \\ 0000\ 0000\ 1100 \\ 0000\ 0000\ 0110 \\ 0000\ 0000\ 0011 \\ 0001\ 0000\ 0011 \\ 0001\ 1000\ 0011 \\ 0001\ 1100\ 0011 \\ 0001\ 1110\ 0011 \\ 0001\ 1111\ 0011 \end{pmatrix} \cdot \begin{pmatrix} \text{stretch}_0 \\ \text{stretch}_1 \\ \text{stretch}_2 \\ \text{stretch}_3 \\ \text{stretch}_4 \\ \text{stretch}_5 \\ \text{stretch}_6 \\ \text{stretch}_7 \\ \text{stretch}_8 \\ \text{stretch}_9 \\ \text{stretch}_{10} \\ \text{stretch}_{11} \end{pmatrix}$$

Figure 31. Scrambler Matrix

Finally, the overlapping streams of two of the four groups can be used to define the other two; by combining specific eight byte stretches i.e., multiplying the stream with a static matrix. For example, to obtain the first stretch of the overlapping stream of SCRMSEEDs `0x4`, `0x10`, `0x100`, `0x1000`, and `0x10000` we combine the fifth and the sixth stretch of the overlapping stream of SCRMSEEDs `0x1`, `0x40`, `0x400`, and `0x4000`. That is `20 00 10 00 08 00 04 00` = `00 01 00 00 00 00 00 00` ^ `20 01 10 00 08 00 04 00`. The matrix is the same between the two groups and provided in Figure 31. One is invited to verify the correctness of that figure using Figure 32.

Some future work remains to be done. We postulate the existence of a mathematical basis to these observations, but a nice mathematical relationship underpinning the observations is yet to be found. Any additional details can be found in my TUE thesis.[60]

[60] `unzip pocorgtfo18.pdf heijningen-thesis.pdf`

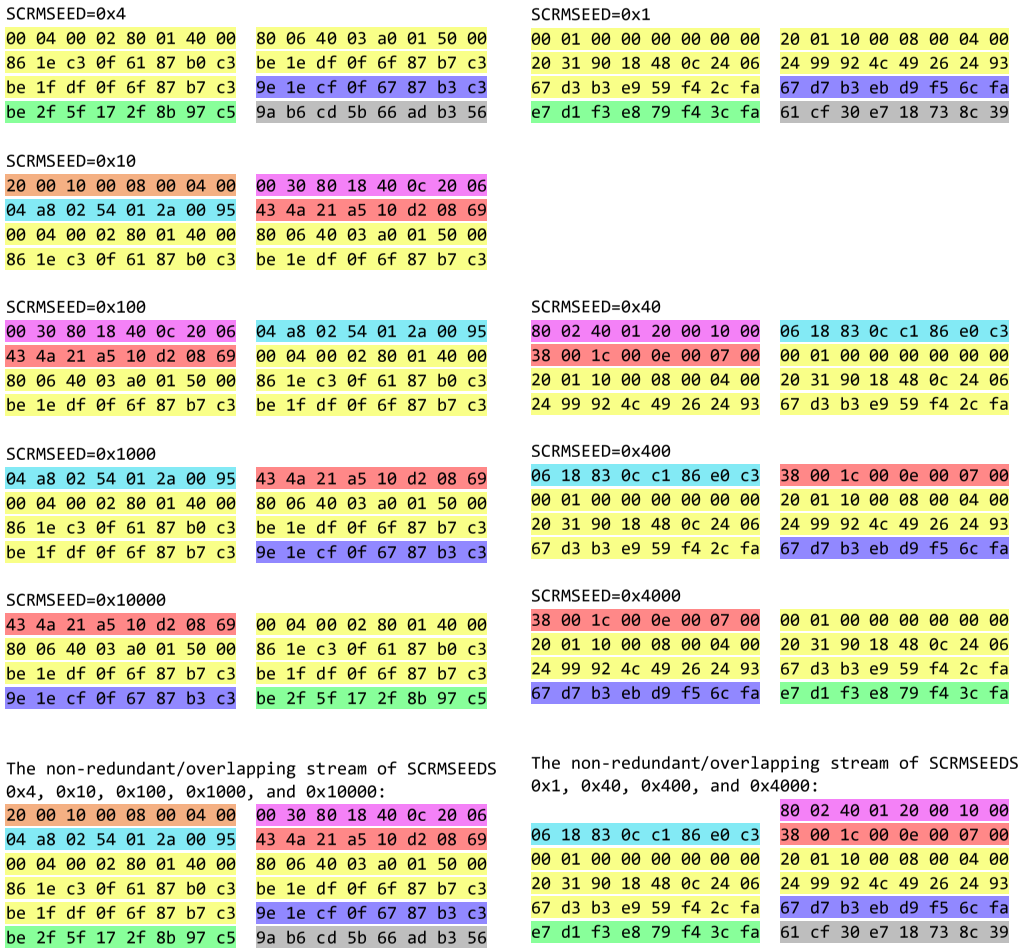


Figure 32. Overlapping Streams