Notes on Intel Microcode Updates

Ben Hawkes <hawkes@inertiawar.com>
December, 2012 - March, 2013

html - pdf

Introduction

All modern CPU vendors have a history of design and implementation defects, ranging from relatively benign stability issues to potential security vulnerabilities. The latest CPU errata release for second generation Intel Core processors describes a total of 120 "erratums", or hardware bugs. Although most of these errata bugs are listed as "No Fix", Intel has supported the ability to apply stability and security updates to the CPU in the form of microcode updates for well over a decade*.

Unfortunately, the microcode update format is undocumented. Researchers are currently prevented from gaining any sort of detailed understanding of the microcode format, which means that it is impossible to study the updates to clearly establish whether any security issues are being fixed by microcode patches. The following document is a summary of notes I gathered while investigating the Intel microcode update mechanism.

** The earliest Intel microcode release appears to be from January 29, 2000. Since that date, a further 29 distinct microcode DAT files have been released.*

Acknowledgements

The initial idea to study Intel's microcode update mechanism was inspired directly from Tavis Ormandy's exploratory work on this subject in 2011. Furthermore, I'd like to thank Emilia Kasper, Tavis Ormandy, Gynvael Coldwind and Thomas Dullien for their outstanding technical assistance and encouragement.

How does the microcode update mechanism work?

Microcode updates are applied to a CPU by writing the virtual address of the Intel-supplied undocumented binary blob to a model-specific register (MSR) called IA32_UCODE_WRITE. This is a privileged operation that is normally performed by the system BIOS at boot time, but modern operating system kernels also include support for applying microcode updates.

The BIOS (or operating system) should verify that the supplied update correctly matches the running hardware before attempting the WRMSR operation. In order to do so, each microcode update comes packaged with a short header containing various update metadata. The header is documented by Intel in Volume 3 of the Developer's Manual. It contains three pieces of information required for validation: the microcode revision, processor signature, and processor flags.

The microcode revision is an incremental version number - you can only successfully apply an update if the current microcode revision is less than the revision supplied. The BIOS will typically extract the current microcode revision by issuing a RDMSR called IA32_UCODE_REV and then compare this value against the revision contained in the new microcode update's header.

The processor signature is a unique representative of the hardware model that the microcode will apply to. The signature of the running hardware can be retrieved using the CPUID instruction, and then compared against the value supplied in the microcode header. According to Intel, "each microcode update is designed specifically for a given extended family, extended model, type, family, model, and stepping of the processor.". The processor flags field is similar, Intel says: "the BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor."

Once a microcode update has been applied using IA32_UCODE_WRITE, the BIOS will typically issue a CPUID instruction and then read the IA32_UCODE_REV MSR again. If the revision number has increased, the update was applied successfully.

Observation #1 - What does a microcode update look like?

Since 2008 Intel has regularly released DAT files containing the most up to date microcode revisions for each processor. Prior to this, microcode update data was shipped as part of the open source tool microcode_ctl. An archive of all microcode DAT releases can be found here.

So what does the undocumented blob portion of the microcode update look like? It appears that there is at least two different formats to the undocumented blob, the old format being used up until Pentium 4 and certain early models of the Intel Core 2, and the new format used from that point onwards. This article covers the new style format only.

The follow graphic shows a microcode update for an Intel Core i5 M460 (i.e. with the documented microcode header stripped):

```
0x00000000,   0x000000a1,   0x00020001,   0x00000003,
0x00000000,   0x00000000,   0x20110831,   0x000001a1,
0x00000001,   0x00020655,   0x00000000,   0x00000000,
0x00000000,   0x00000000,   0x00000000,   0x00000000,
0x00000000,   0x00000000,   0x00000000,   0x00000000,
0x00000000,   0x00000000,   0x00000000,   0x00000000,
0xc3323f9e,   0xa2912832,   0xa7a30a2e,   0x9e75a181,
0x840159be,   0xd1c7dbb0,   0x921c5cf7,   0xb323111b,
0x5d5f57b5,   0x9926d84a,   0xdaaf70e7,   0x91786b18,
0x82da740d,   0xae90be68,   0x8b6a07cf,   0xc2c37ebc,
0xde451f9c,   0x7a1444c0,   0xce6700b1,   0x5c29e58a,
0x12eb603f,   0xaaf6cb53,   0xd807a1c9,   0xf51ed696,
```

fig 1 - initial 192 bytes of undocumented microcode update data for Intel Core i5 M460

It is immediately clear that there is a plaintext structure (96 bytes in length) at the start of the undocumented blob. Some easily identifiable fields are colorized:

- `0x00000003` Microcode revision number.
- `0x20110831` Release date (note that this date is sometimes one day prior to the microcode header date).
- `0x000001a1` Real length of microcode update (counted in 4-byte words).
- `0x00020655` Processor signature.

And some less easily identifiable fields that appear to be in common usage are marked in grey:

- `0x00000000` Possible flags field? May not be in use in recent hardware types.
- `0x00000001` Possible loader version?
- `0x00000000` Possible length field (when non-zero)? Not consistently used.

Observation #2 - Is there any structure in the microcode update after the 96 byte header?

Most of the data located after the 96 byte header appears to be random and without structure. However, performing a longest common substring analysis on an archive of every unique microcode update (available in binary format here) showed that different revisions for the same (or similar) processor signatures will share some common byte strings:

```
Revision 2 for signature/pf 0x20655/0x800
0x7b6d549d,   0xb6a6a8ec,   0x36daa74b,   0xf9301a9f,
0x5313aa68,   0xf218b6ec,   0x6a6bea7c,   0x01449bf7,
0x5d5f57b5,   0x9926d84a,   0xdaaf70e7,   0x91786b18,
0x82da740d,   0xae90be68,   0x8b6a07cf,   0xc2c37ebc,
0xde451f9c,   0x7a1444c0,   0xce6700b1,   0x5c29e58a,
0x12eb603f,   0xaaf6cb53,   0xd807a1c9,   0xf51ed696,
0x90d9e2c3,   0x641bff09,   0x732ab820,   0xfef7aa2f,
0x8f518e1d,   0xf0d8c4d9,   0x73aefe19,   0xf6d7ae12,
0xd50a095b,   0xdf21cbb8,   0x4b81ff9a,   0xe1a6d1f8,
0xbe78b0de,   0xda86826f,   0xc4f9c057,   0x306f1761,
0x7aae81f0,   0xdbab0ae2,   0x789ec0ff,   0xbaacd096,
0x68fa4733,   0x2f3fd406,   0xd287247f,   0x79eefe4c,
0x6f674a41,   0xd6948ad8,   0x90388682,   0x5a4bf35d,
0xadab41ff,   0x89e9dc62,   0xb2eccfee,   0xd4d52153,
0x34301d19,   0x8a2f38a6,   0x35b05e56,   0x482f029a,
0x296cb5a3,   0xce64f437,   0xae1dfa06,   0xb9f29132,
0x1960a103,   0x70bb6f46,   0x152a5d83,   0x171a19d9,
0xac9aa4b7,   0x54b719f7,   0x5b781cb1,   0xaea548bb,
0x00000011,   0xc1bc7c29,   0xeb21694f,   0x72a42f1d,
0xa5771d59,   0xc9c2f00e,   0x02335972,   0xc046f033,
0x9d039c23,   0x146b14bd,   0xdcd21dbf,   0xf72c8a2a,

Revision 3 for signature/pf 0x20655/0x800
0xc3323f9e,   0xa2912832,   0xa7a30a2e,   0x9e75a181,
0x840159be,   0xd1c7dbb0,   0x921c5cf7,   0xb323111b,
0x5d5f57b5,   0x9926d84a,   0xdaaf70e7,   0x91786b18,
0x82da740d,   0xae90be68,   0x8b6a07cf,   0xc2c37ebc,
0xde451f9c,   0x7a1444c0,   0xce6700b1,   0x5c29e58a,
0x12eb603f,   0xaaf6cb53,   0xd807a1c9,   0xf51ed696,
0x90d9e2c3,   0x641bff09,   0x732ab820,   0xfef7aa2f,
0x8f518e1d,   0xf0d8c4d9,   0x73aefe19,   0xf6d7ae12,
0xd50a095b,   0xdf21cbb8,   0x4b81ff9a,   0xe1a6d1f8,
0xbe78b0de,   0xda86826f,   0xc4f9c057,   0x306f1761,
0x7aae81f0,   0xdbab0ae2,   0x789ec0ff,   0xbaacd096,
0x68fa4733,   0x2f3fd406,   0xd287247f,   0x79eefe4c,
0x6f674a41,   0xd6948ad8,   0x90388682,   0x5a4bf35d,
0xadab41ff,   0x89e9dc62,   0xb2eccfee,   0xd4d52153,
0x34301d19,   0x8a2f38a6,   0x35b05e56,   0x482f029a,
0x296cb5a3,   0xce64f437,   0xae1dfa06,   0xb9f29132,
0x1960a103,   0x70bb6f46,   0x152a5d83,   0x171a19d9,
0xac9aa4b7,   0x54b719f7,   0x5b781cb1,   0xaea548bb,
0x00000011,   0x649668c5,   0xd3d4e73c,   0x0d7eee17,
0xc8a4d8f1,   0xa531e67c,   0x0d4e93f9,   0x109a2c8e,
0xb3a120eb,   0x1cb99ad0,   0xdf7fffa7,   0xcd22ae67,
```

fig 2 - comparison of the section of bytes after the 96-byte header for two different microcode revisions for Intel Core i5 M460

In this figure, two distinct strings have been identified:

- In green, a 2048-bit string that is constant between microcode revisions.
- In red, a 32-bit string that is constant for all microcode updates using the new style format.

In total, 12 unique 2048-bit strings were found to be shared across 24 processor signatures. The extracted data is available here (in the format <2048-bit string>).

Note that 2048-bits is a commonly used length for an RSA modulus, and that 0x00000011 (decimal 17) is a commonly used value for an RSA exponent. This suggests that these common strings may be an RSA public key. Further evidence to support this claim is that:

- Each of the values are strictly 2048 bit in length, i.e. the most significant bit is always set.
- None of the values are trivially factorable by 2, i.e. the values are all odd numbered.
- None of the values are factorable by any value between 2 and 2^32.

Observation #3 - Can the length of the microcode update be verified?

The length field of the 96-byte microcode header (shaded in green in fig 1) can be verified using a fault injection analysis. The idea is to sequentially mutate each byte of a valid microcode update, attempt to apply the update, and record whether the update was applied successfully or not.

The underlying assumption here is that the CPU should validate the integrity of the microcode update, but may not validate the integrity of padding (since microcode updates must be a multiple of 1024, it is assumed that padding is normally required).

Testing on an Intel Core i5 M460 (sig 0x20655, pf 0x800), the expected length of the microcode update (in revision 3) is 1668 bytes (0x1a1 * 4). Sequentially flipping a bit in each byte from offset 0 to 2000 and waiting for the first successfully applied update gives the following results:

```
...
Dec  2 14:47:54 rhyme kernel: [   21.002342] DH V5 MOD OF OFFSET 1660 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.002614] DH V5 MOD OF OFFSET 1661 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.002887] DH V5 MOD OF OFFSET 1662 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.003159] DH V5 MOD OF OFFSET 1663 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.003431] DH V5 MOD OF OFFSET 1664 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.003703] DH V5 MOD OF OFFSET 1665 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.003976] DH V5 MOD OF OFFSET 1666 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.004249] DH V5 MOD OF OFFSET 1667 CAUSED UPDATE FAILURE
Dec  2 14:47:54 rhyme kernel: [   21.004559] DH V5 MOD OF OFFSET 1668 HAD NO EFFECT
```

fig 3 - output of kernel module mutating at each byte offset from zero, applying microcode update and waiting for success

This result was observed on Intel Core 2 Duo P9500, Intel Core i5 M460 and Intel Core i5 2520M chips. For all other experiments below, results were reproduced on Intel Core i5 M460, Intel Core i5 2520M, and Intel Xeon W3690 chips.

Observation #4 - How many cycles does an update take to be applied successfully?

To collect the average number of cycles the CPU took to successfully apply a microcode update, a specialized system was setup that would:

1. Boot the system with an initial microcode revision.
2. Install a Linux kernel module that:
    a. Invalidate caches (wbinvd)
    b. Stop instruction prefetch (sync_core)
    c. Disable interrupts for the running core (local_irq_disable)
    d. Record time stamp counter (rdtsc)
    e. Apply the next microcode update revision (wrmsr MSR_IA32_UCODE_WRITE)
    f. Record time stamp counter

1. Record time stamp counter
3. Record the rdtsc delta in syslog
4. Reboot

The cache invalidation and interrupt disable were intended to reduce variance in the timing delta. Rebooting is required to reset the system to the original microcode revision, as successfully applied revisions must be strictly incremental.

The exact cycle value will vary significantly between different types of hardware (older hardware was observed to take significantly more cycles), however a baseline value can be used in further timing analysis on the same hardware. For example, the baseline average time delta across 2000 applications of microcode revision 3 for an Intel Core i5 M460 is:

Average: 488953 cycles
Sample standard deviation: 12270 cycles

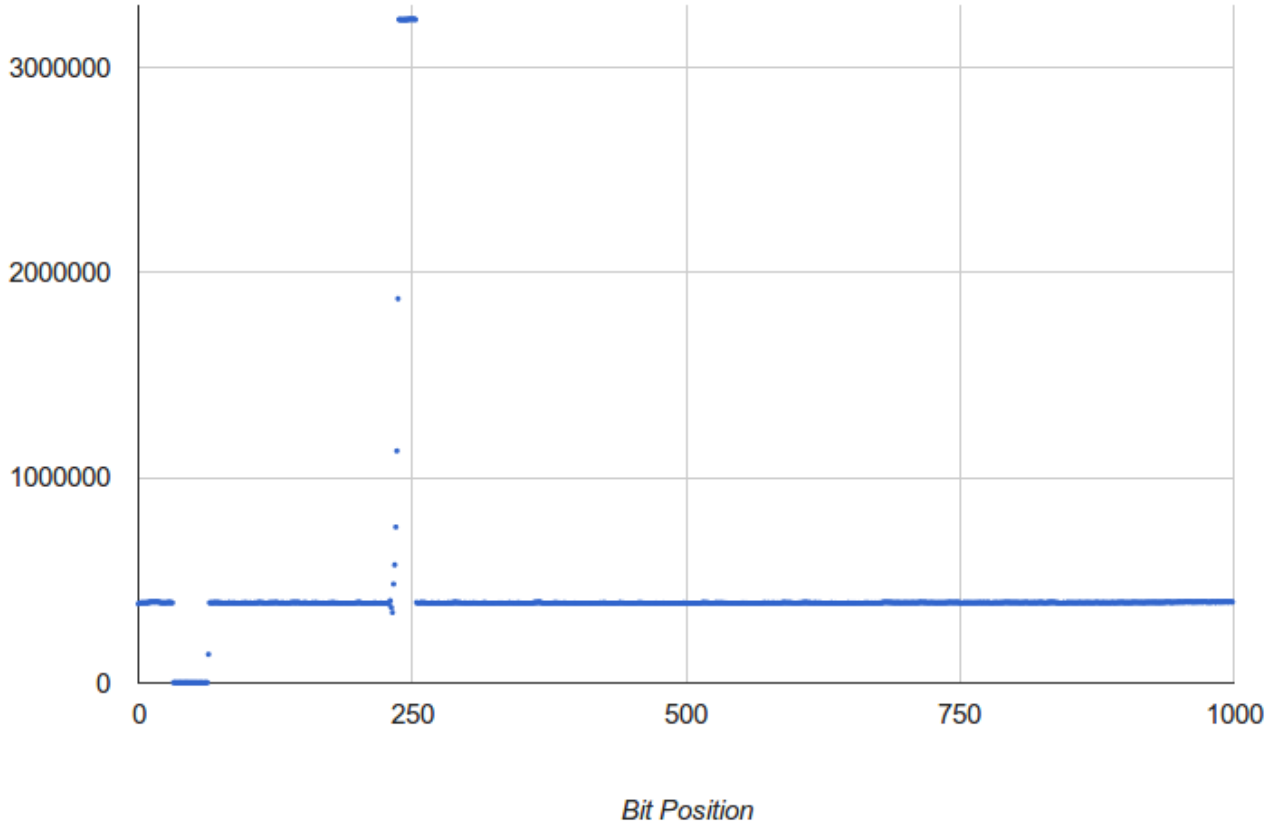
The high variation in the sample deltas collected is presumed to be caused by multi-core systems. If the microcode update mechanism has to achieve a consistent state across all available instruction pipelines (including consistency across hyperthreads, prefetched instructions, instruction caches on all cores), this could result in a high level of variance, as the collection mechanism used here only "cleans" internal state for the running core.

Observation #5 - Do the number of cycles change depending on the location of a fault?

Using the baseline timing delta above, it is possible to find deviations by flipping every possible bit position in the microcode update and attempting to apply the malformed update. All of these update attempts will fail, but the idea is that certain fields may be treated differently by the microcode update mechanism, and that this may show up in the cycle delta.

Running this test on an Intel Core i5 M460 gives the following results:

Fig 4 - Fault Injection Timing Analysis



This chart shows the results of the first 1000 bit positions being flipped. Three distinct areas of interest can be seen. All other bit position above 1000 return a cycle count matching the failure case seen above.

The first area of interest, between bit offsets 32 and 63, corresponds to an unknown word the in the 96-byte header that always has value 0x000000a1. This may serve as a magic value, checked when the microcode is first loaded to ensure that an expected format has been received.

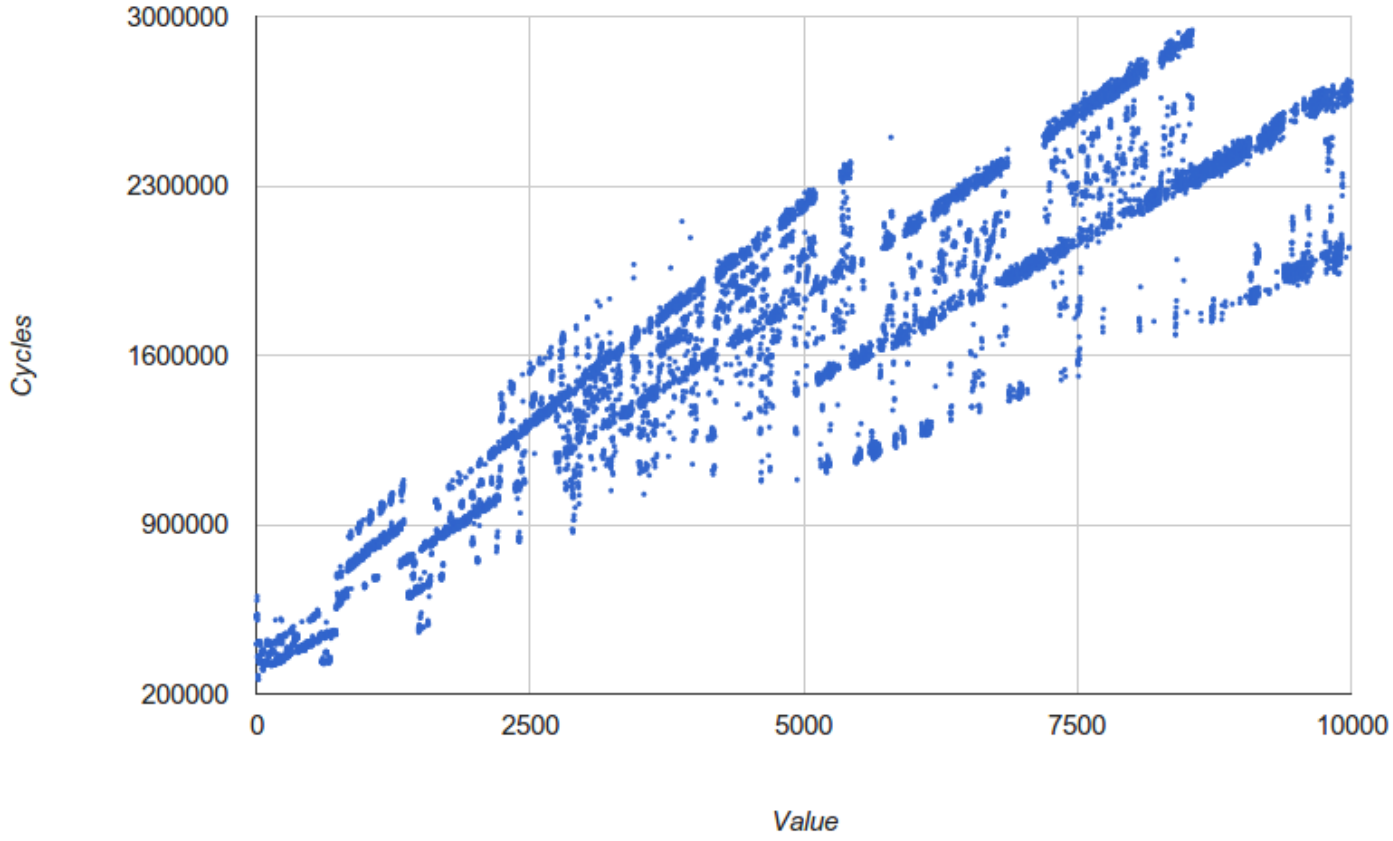
The second area of interest is a single bit at offset 64, which appears to correspond to a flags field. In the original analysis, this bit was set. However, clearing the bit and repeating the analysis shows identical results to figure 4, except with a significantly lower average count of cycles for the "normal" failure case. The decrease in cycle count appears to be proportional to the number of physical cores on the system, which may suggest this bit is used to decide whether the update will be iteratively applied to all cores, or only applied on a single core.

The third area of interest, between bit offsets 233 and 253, corresponds to the microcode size field.

Observation #6 - What happens of the microcode size field is modified?

Modifying each bit position results in an incrementally higher cycle count. To investigate this further, a second analysis was run that records the cycle count for each size value between 0 and 10000. The following shows the results of this analysis on an Intel Core i5 2520M:

Fig 5 - Microcode Size Modifcations

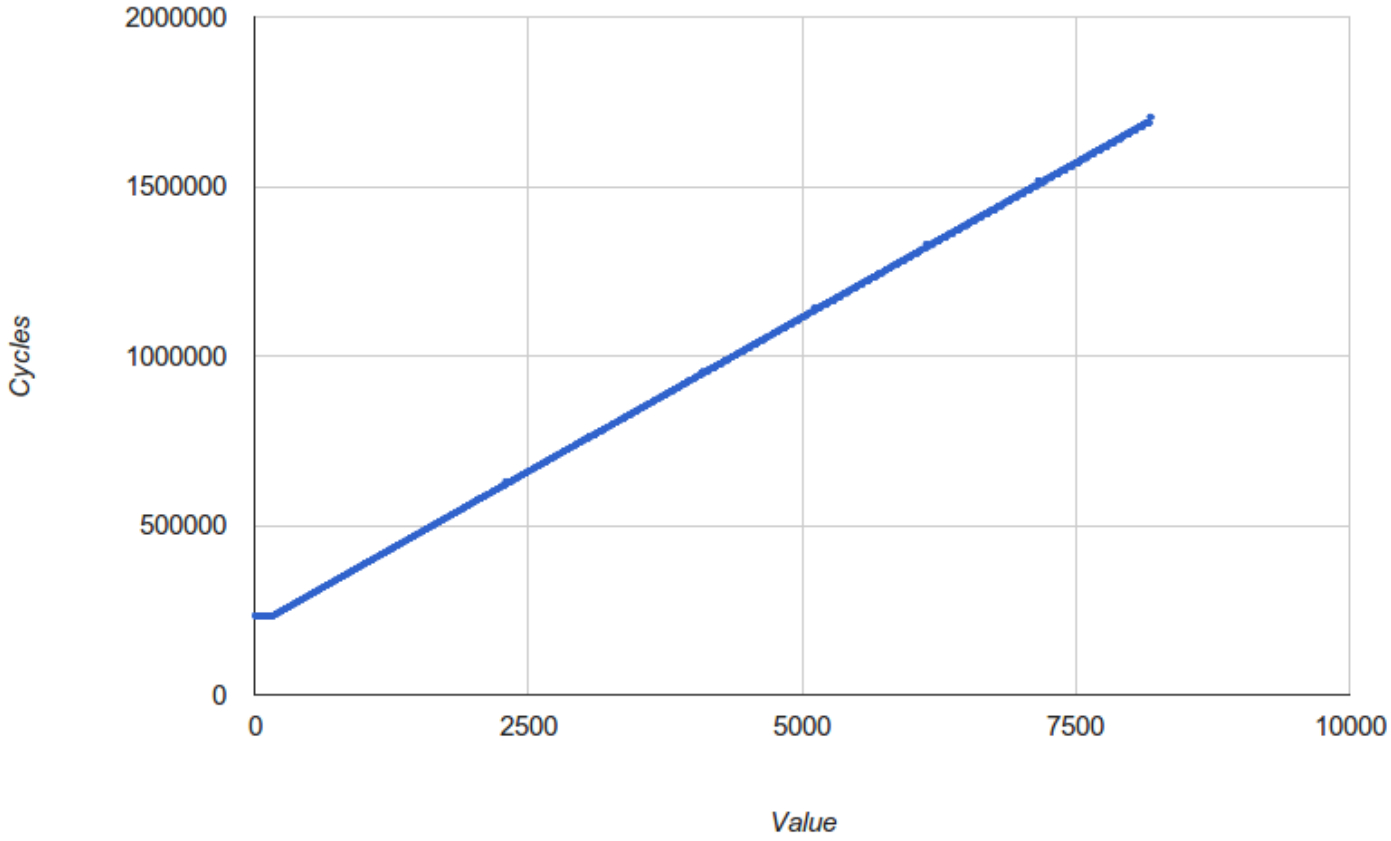


In this chart we can see a clear correlation between an increasing size value and an increasing cycle count. This chart appears to also show artifacts from running this system on a multi-core system (note that the i5 2520M is a quad core processor, and that four main trend lines can be seen).

Running the size modification analysis with an incorrect magic value (i.e. replacing 0x000000a1 with a different value) results in a flat chart with no correlation between value and cycle count. This suggests that the magic value is checked prior to the size value being used.

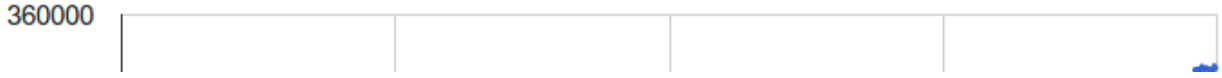
Due to the high level of noise while running this analysis on a multi-core system, the analysis was rerun with symmetric multiprocessing (SMP) and HyperThreading disabled. A clear linear correlation between length value and cycle count is seen. The follow data is taken from an Intel Core i5 2520M:

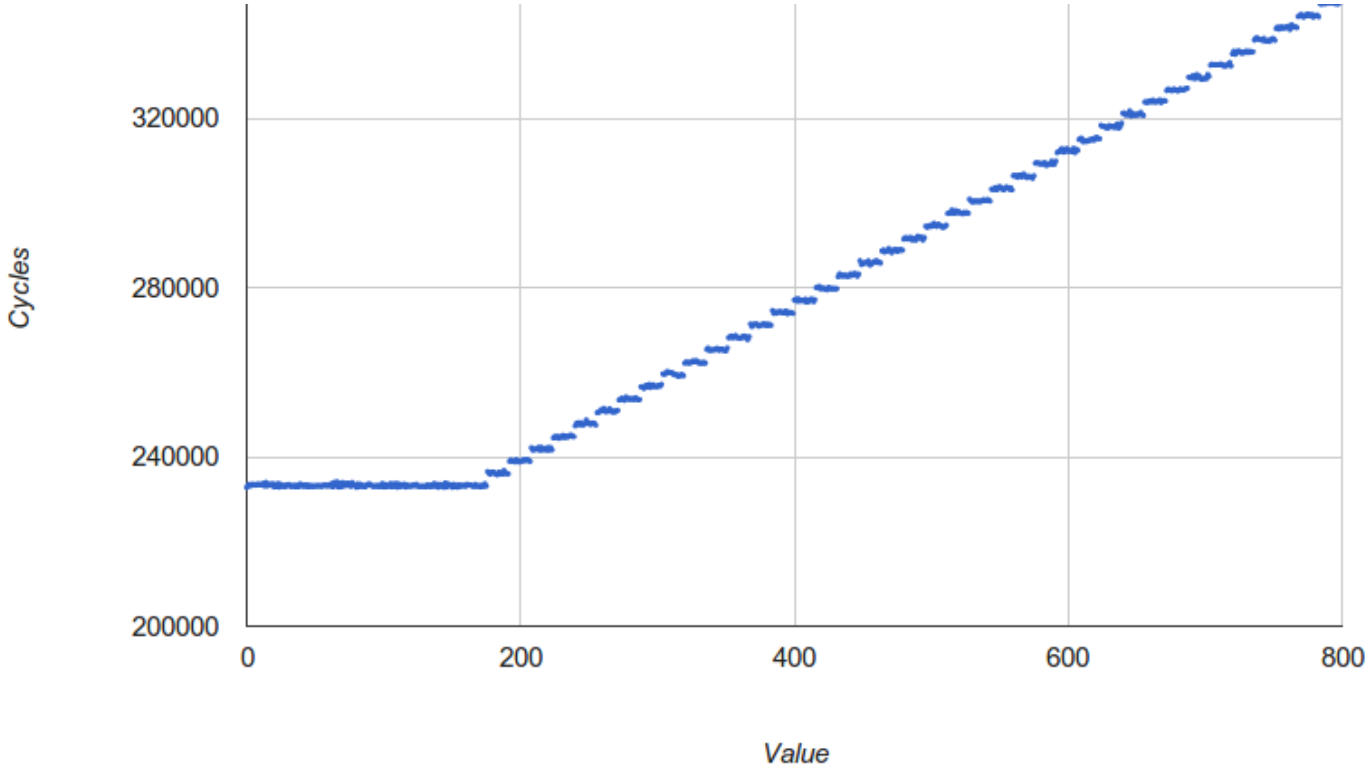
Fig 6 - Microcode Size Modifications (single core)



With this cleaner data, it is possible to observe new timing behavior. By displaying a smaller sample, clear timing shelves are seen as the size value increases:

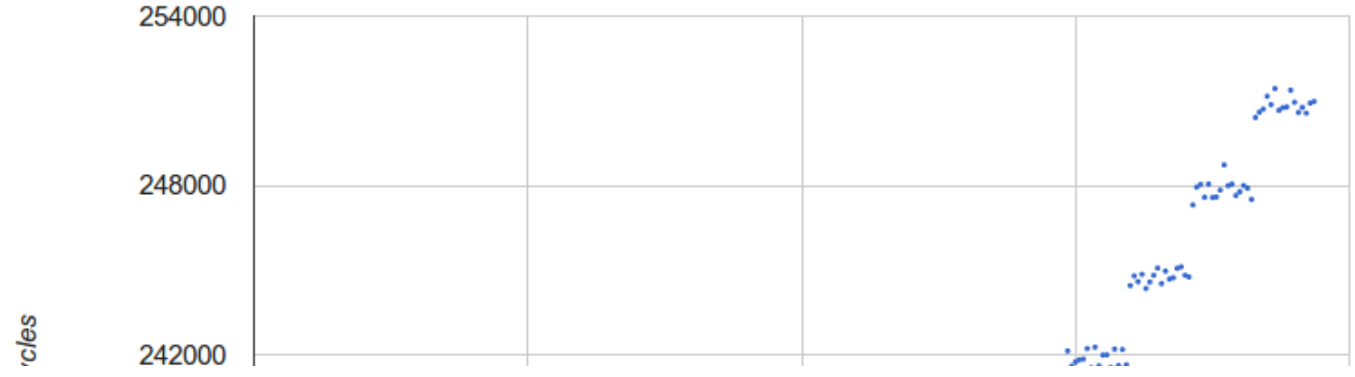
Fig 7 - Microcode Size Modifications (single core)

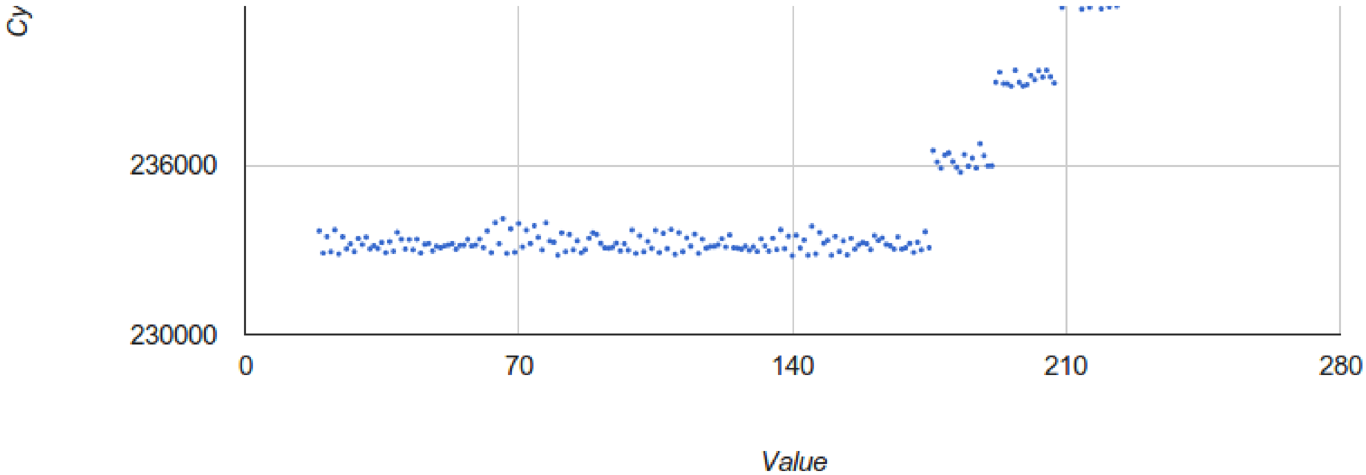




By observing the individual points of the timing shelves, it is clear that each timing shelf has 16 points. Since each single increase in size value corresponds to a 4-byte increase in microcode data, 16 points represents 512 bits of data.

Fig 8 - Microcode Size Modifications (single core)





512-bits is the standard message block size for popular cryptographic hash functions such as MD5, SHA1 and SHA2. The timing shelves observed match what we would expect from a Merkle-Damgard hash function, as each new shelf represents the increase number of cycles required to process a new message block.

In public-key signature schemes, it is normal to sign a hash of the data message instead of signing the entire message contents. This means that a hash operation being observed in the early stages of the microcode loader process is an expected result.

The lack of timing artifacts corresponding to symmetric key algorithm block sizes (i.e. 128-bits) may also indicate that authentication of the microcode contents is occurring prior to decryption of the microcode contents (i.e. the cipher-text is authenticated). Given the space constraints of a modern CPU architecture, this design is not entirely unexpected, as it allows the processor to load the decrypted content directly, without having to store the plaintext for authentication purposes.

Observation #7 - What other data is in the first 704 bytes of a microcode update?

Note that the first shelf is observed after supplying a size value of 176 (or 704 bytes of microde data), and that supplying a size value of 704 bytes or less results in a constant timing shelf. This would suggest that there exists a minimum length of non-variable-length data that will be hashed regardless of the supplied microcode size field. This data includes the undocumented microcode header and the RSA public key that has been discussed above.

If we assume that the presence of an RSA public key suggests the usage of RSA as a digital signature algorithm, then it stands to reason that an RSA signature will be found in the microcode update. If this signature value is calculated using the public key embedded in the microcode update, then we would expect to find a 2048-bit value that is strictly less than the modulus value (since the signature is calculated using this modulus).

Examining the 2048-bits that are contiguously after the public key exponent value (0x00000011), we find a valid candidate for an RSA signature. In every case, the 2048-bit value after the exponent is strictly less than the 2048-bits prior to the exponent (the presumed RSA modulus).

We can attempt to recover the originally signed data by raising the signature value to the power of 0x00000011 and then using the modulus value. The results of this operation can be found here. The format of this file is <processor signature> <microcode version> <result>.

The result appears to use PKCS#1 v1.5 padding, with a private-key operation set for the block type. It is also clear that earlier processor models used a 160-bit digest for the signature hash, which is consistent with SHA1. Later processor models use a 256-bit digest, which is consistent with SHA2.

```
1fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffff002dcf8295927a6f32e573e3bac70e5f8c3ecf600f
```

```
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00be7346f75de
e9ca4977fc717e7aa10d5a45ebbf274193a8f92fe4582dba562a4
```

fig 9 - Comparing signed data on older / newer processor models (160-bits marked in blue, 256-bits marked in red)

All attempts at recreating these hash values using standard SHA implementations have failed. Several non-standard variations of Merkle-Damgard strengthening were also attempted. This may indicate that a non-standard initial vector or some other non-standard structural variation is used when calculating the signed hash value.

Attempts to insert a new public key and signature for the same PKCS#1 signed data into the microcode also failed, which suggests that the public key is part of the authenticated data, or that a hash of the expected/official public key is stored in factory embedded memory and verified after authentication.

Interestingly, it was observed that setting the most significant word of the public key modulus to zero results in a hardware reset (in the case of a single core system, this manifests as a hardware halt/freeze, not a system restart). This may suggest a "division by zero" error exists in the microcode authentication routine.

Conclusion

Studying the Intel microcode update mechanism through data analysis and timing analysis has revealed properties about the cryptographic design of this system:

- Several previously undocumented header fields have been identified and described.
- The results suggest that microcode updates are authenticated using a 2048-bit RSA signature.
- The RSA signature operation appears to be constant-time (i.e. unaffected by changes to the supplied exponent, modulus or signature value).
- Timing analysis reveals 512-bit steps correlating to supplied microcode length. This is a common message block size for cryptographic hash functions such as SHA1 and SHA2
- The RSA signature was located, and the signed data is a PKCS#1 1.5 encoded hash value. Older processor models use a 160-bit digest (SHA1), and newer process models use a 256-bit digest (SHA2).