BLOG <!--enterBlog-- >

Zen and the Art of Microcode Hacking

Josh Eads
Information Security Engineer

Tavis Ormandy
Software Engineer

Matteo Rizzo
Information Security Engineer

Kristoffer Janke
Information Security Engineer

Eduardo Vela Nava

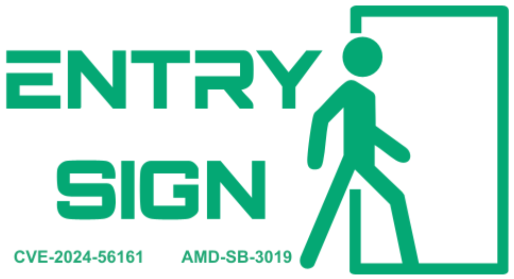


Fig. 1. The EntrySign vulnerability logo (CC0)

Today we are releasing the full details of EntrySign, the AMD Zen microcode signature validation vulnerability which we initially disclosed last month.

In this post, we first discuss the background of what microcode is, why microcode patches exist, why the integrity of microcode is important for security, and how AMD attempts to prevent tampering with microcode. Next, we focus on the microcode patch signature validation process and explain in detail the vulnerability present (using CMAC as a hash function). Finally, we discuss how to use some of the tools we've released today which can help researchers reproduce and expand on our work (skip to the Zentool section of this blogpost for a “how to” on writing your own microcode).

In a future post, we will walk through the methods and techniques used to discover this vulnerability along with further details on the microcode patch and instruction format.

Modern x86 CPUs consist of many complex instruction set computer (CISC) cores, each of which internally uses a reduced instruction set computer (RISC), termed the *microcode engine*, to implement some of the more complicated instructions and architectural transitions. Both Intel and AMD designed unique RISC-based microcode instruction sets, both of which are undocumented, but conceptually similar to other RISC instruction sets like ARM or RISC-V. Similar to software, implementing complex hardware correctly is challenging; historically, this has led to multiple well-known bugs, such as the Intel FDIV bug on Pentium in the year 1994. Unlike software, when a bug is discovered in hardware oftentimes the only way to remediate the issue is to fabricate a new, fixed version of the hardware – an extremely costly process. To avoid this massive cost, x86 CPU manufacturers created a mechanism enabling them to update the CPU's microcode at runtime in order to patch known bugs (available from AMD K8 in 2003, and Intel P6 in 1995).

The diagram below depicts a simplified view of the AMD Zen CPU architecture with an emphasis on the sections involving microcode. Similar to other CPU architectures, at a high level the core is split into a frontend which fetches and decodes instructions, and a backend which executes and retires instructions. Once the current instruction pointer address is determined, the CPU checks to see if the micro-ops implementing the instruction are present in the micro-op cache; if so, then those micro-ops are pushed into the queue to be dispatched to the backend. Otherwise, the instruction cache is queried which eventually (depending on data locality) will return back the bytes at the instruction pointer which encompass the x86 macro instruction. From here, the decoder determines whether the instruction is a "fastpath" or microcoded instruction. Fastpath instructions are implemented using a hard-coded set of micro-ops, while microcoded instructions require the microcode engine to emit a variable length set of micro-ops. Note the patch RAM adjacent to the microcode ROM –

micro-op queue and sent to the dispatcher (similar to the micro-op cache hit path from before).

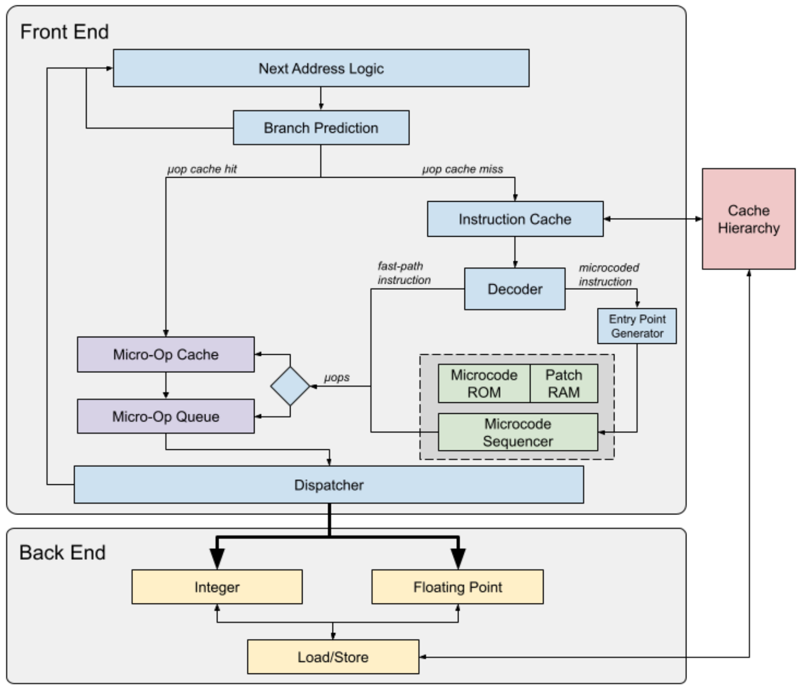


Fig. 2. A simplified view of AMD's Zen architecture highlighting where the microcode engine might be

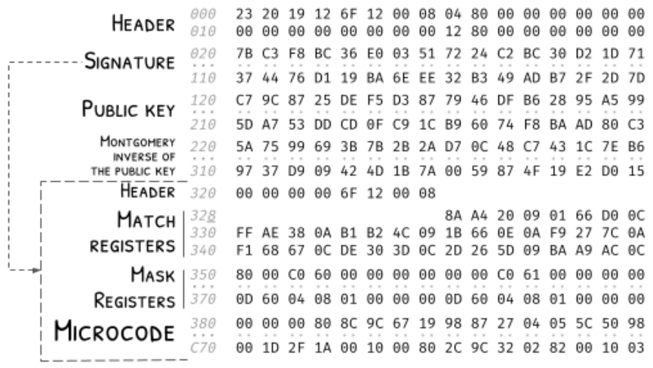
The CPU microcode instructions and data are physically stored in an on-die ROM with additional on-die patch SRAM. This patch RAM can be loaded with new instructions during a microcode update (typically occurring during BIOS and OS boot) and is impossible to directly access using the x86 core. The only documented method for changing the microcode patch RAM is by loading an authentic microcode update from AMD, verified by a cryptographic signature.

AMD Microcode Patch Routine

Microcode patching provides CPU architects with an amazing amount of flexibility when new hardware bugs are encountered and need to be fixed. However, Intel and AMD have utilized encryption to prevent reverse engineering of patches and digital signatures to prevent and loading process of AMD microcode patches comprises 4 steps and is detailed below:

I) Authorship: AMD generates a new microcode patch. The patch is delivered as a binary blob to BIOS, OS, and other partners; inside this blob are the following components:

- A header describing the patch header format, metadata about the CPU it is designed for, date, and versioning information.
- A 2048-bit RSA PKCS #1 signature.
- A 2048-bit RSA public key modulus (0x10001 is used as the exponent).
- A 2048-bit Montgomery inverse of the public key (used to simplify RSA modular operations).
- A bit which indicates whether the remainder of the patch is encrypted.
- An array of match registers and mask values which select which microcode and instructions to patch.
- An array of micro-ops bundled in sets of four “quads”, each paired with a sequence word indicating where to execute next.



```
000   23 20 19 12 6F 12 00 08 04 80 00 00 00 00 00 00
010   00 00 00 00 00 00 00 00 12 80 00 00 00 00 00 00
020   7B C3 F8 BC 36 E0 03 51 72 24 C2 BC 30 D2 1D 71
...
110   37 44 76 D1 19 BA 6E EE 32 B3 49 AD B7 2F 2D 7D
120   C7 9C 87 25 DE F5 D3 87 79 46 DF B6 28 95 A5 99
...
210   5D A7 53 DD CD 0F C9 1C B9 60 74 F8 BA AD 80 C3
220   5A 75 99 69 3B 7B 2B 2A D7 0C 48 C7 43 1C 7E B6
...
310   97 37 D9 09 42 4D 1B 7A 00 59 87 4F 19 E2 D0 15
320   00 00 00 00 6F 12 00 08
328                           8A A4 20 09 01 66 D0 0C
330   FF AE 38 0A B1 B2 4C 09 1B 66 0E 0A F9 27 7C 0A
340   F1 68 67 0C DE 30 3D 0C 2D 26 5D 09 BA A9 AC 0C
350   80 00 C0 60 00 00 00 00 00 00 C0 61 00 00 00 00
...
370   0D 60 04 08 01 00 00 00 0D 60 04 08 01 00 00 00
380   00 00 00 80 8C 9C 67 19 98 87 27 04 05 5C 50 98
...
C70   00 1D 2F 1A 00 10 00 80 2C 9C 32 02 82 00 10 03
```

Fig. 3. An example of a microcode patch file

II) Authentication: AMD signs the new microcode patch using their RSA private key which corresponds to the public key embedded in the patch. Later, during patch verification, the CPU will hash the RSA ensure only the original RSA key pair can be used to sign microcode patches.

III) Delivery: AMD delivers the microcode patch to OEMs, OS platforms, and other partners for validation and distribution.

IV) Verification & Installation: Once the new patch is received, the microcode will be loaded either at runtime or during the next reboot.

1. The BIOS or OS will identify the correct microcode patch file (based on CPU identifiers in the header) and begin the update routine.
2. The software will write the virtual address of the microcode patch blob to MSR 0xc0010020, this instructs the CPU microcode to start executing the microcode update routine.
3. The microcode copies the patch to internal memory and verifies the patch's CPU identifiers match the actual hardware's identifiers.
4. The microcode checks whether the patch's version is older than the currently installed patch version. If so, the patch is rejected – this prevents rollback attacks.
5. The patch hashes the RSA public key using AES-CMAC and confirms the hash matches the value which was fused in by AMD when manufacturing the chip.
6. The patch hashes the patch contents (match registers, instruction masks, and patch instructions).
7. The RSA PKCS #1 signature is decrypted using the RSA public key and supplied Montgomery modular inverse. The result is a padded AES CMAC hash of the patch contents.
8. If the signed hash matches the calculated hash, the patch contents are copied into internal CPU patch RAM. Otherwise, the patch is rejected.
9. The CPU microcode patch version (MSR 0x8b) is updated to reflect the newly installed patch.

Verification Algorithm

AMD Zen CPUs use an almost standard RSASSA-PKCS1-v1_5 algorithm; however, instead of using one of the recommended hash functions, an alternative that is prone to collisions was selected.

RSA is a classic public-key cryptosystem and RSASSA-PKCS1-v1_5 is a digital signature scheme constructed from RSA. The way standard RSA signature schemes work is by selecting a hash algorithm, like SHA-256, padding it with a constant value (like 0x01FF ... FF00 and a hash function identifier) and then calculating the signature of the padded value. The recipient can then use the public key to validate the signature; essentially encrypting the signature into the padded hash and comparing this value with the calculated padded hash of the contents. *This means that for someone to "fake" a signature, they would need to either break RSA or break the hash function* (usually MD5, SHA-1 or SHA-2).

For situations where minimizing data storage is important (like in a CPU mask ROM or fuses), it is also common practice to store the hash of a public key instead of the public key itself. The AMD microcode patch uses a 2048-bit RSA key, but a 128-bit hash can be stored on-die (on the chip) to more efficiently verify the key is authentic. When verifying a signature, one can hash the public key and verify that it matches the expected value before using it. From a security perspective this is safe and equivalent to matching against the entire RSA key as long as the hash function is secure.

A hash function is a function that takes an arbitrarily long input and produces a fixed output. In the example below, you can see that we have two 6-bit long strings (split in blocks of 3 bits each) that each generate a different 3 bit output.

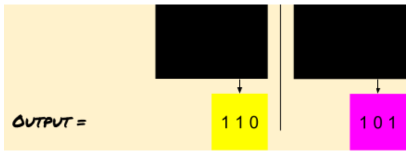


Fig. 4. A hash function takes arbitrarily long input and produces fixed-length output

A secure hash function ensures, among other things, that *given an output it is not feasible for you to reverse the operation and calculate its input* (in crypto terms this is called a ***preimage***). So, for example, given just the output 101, it should be impossible for you to know that 010111 would hash to 101 short of testing all possible inputs until you get the output you wish to calculate by chance. While in this toy example the hash is just 3 bits, so one just has to try a few times (up to eight in the worst case scenario) to find a *preimage*, a typical hash function has an output of at least 128 bits, which under normal circumstances would be too expensive to obtain by chance.

Colliding Keys

The root cause of the EntrySign vulnerability is that the AMD Zen microcode signature verification algorithm uses the CMAC function as a hash function; however, CMAC is a message authentication code and does not necessarily provide the same security guarantees as a cryptographic hash function. A simplified summary of the CMAC algorithm is presented below.

Let's say we want to calculate the 3-bit CMAC of the 6-bit message 000111. First, we have to split the 000111 bit string into blocks (here our example uses blocks of 3 bits, so 000 and 111):

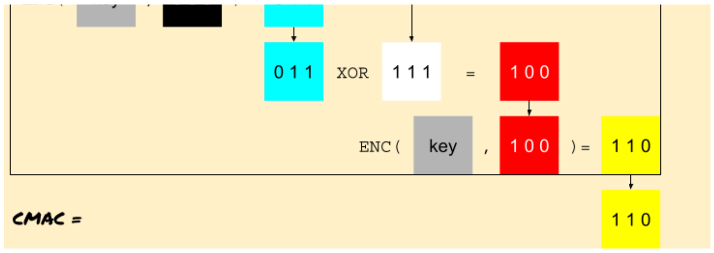


Fig. 5. An example of how the CMAC algorithm works

The way the CMAC algorithm works is simply by using an encryption algorithm (in this case it was AES), and chaining the output with the next block using the XOR operation (note the diagram omits a final step for simplicity, you can read more about how CMAC works here). This ensures that any modification to the input (for example, by changing the plaintext from 000111 into 010111), generates an unpredictable change to the output (from 110 to 101):

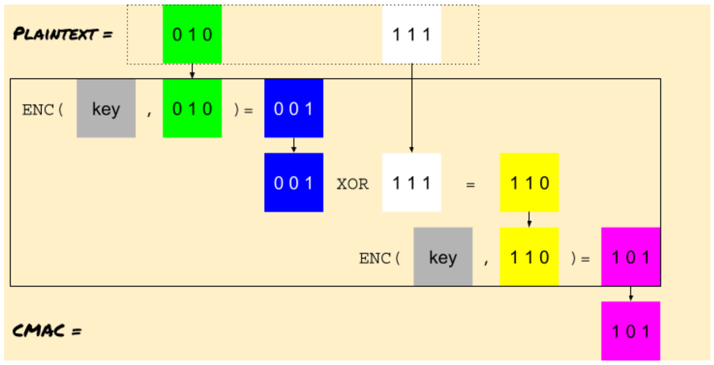


Fig. 6. If the input changes, the CMAC algorithm generates an unpredictable change to the output

The weakness of using CMAC as a hash function is that anyone who has the encryption key is able to observe the intermediate values of the encryption and calculate a way to "correct" the difference so that the final output remains the same, even if the inputs are completely different. For example, below we calculate that given the prefix 010, if we change the second block from 111 to 101, it will make the last operation output the same as it would have been if the input was 000111. Despite the two inputs being completely different, they hash to the same value (110), and we are able to calculate the

100).

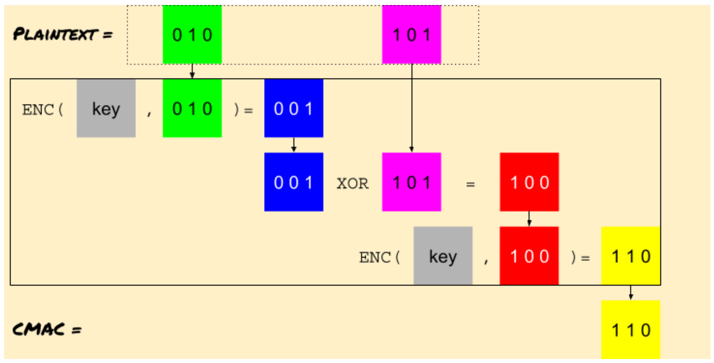


Fig. 7. Manipulating CMAC to receive the desired output value

Secure hash functions are designed in such a way that there is no secret key, and there is no way to use knowledge of the intermediate state in order to generate a collision. However, CMAC was not designed as a hash function, and therefore it is a weak hash function against an adversary who has the key. Remember that every AMD Zen CPU has to have the same AES-CMAC key in order to successfully calculate the hash of the AMD public key and the microcode patch contents. Therefore, the key only needs to be revealed from a single CPU in order to compromise all other CPUs using the same key. This opens up the potential for hardware attacks (e.g., reading the key from ROM with a scanning electron microscope), side-channel attacks (e.g., using Correlation Power Analysis to leak the key during validation), or other software or hardware attacks that can somehow reveal the key. *In summary, it is a safe assumption that such a key will not remain secret forever.*

Forging On

We noticed that the key from an old Zen 1 CPU was the example key of the NIST SP 800-38B publication (Appendix D.1 *2b7e1516 28aed2a6 abf71588 09cf4f3c*) and was reused until at least Zen 4 CPUs. Using this key we could break the two usages of AES-CMAC: the RSA public

Additionally, we calculated collisions for signatures, and were able to generate a microcode patch that shares the same signature as another message that was legitimately signed.

One downside of the attack is that in order to calculate a collision, we had to inject a specific "compensating" block, which looks like random data, and which had to be aligned to 16 bytes. While this usually wouldn't be a problem, doing so on the microcode could cause crashes depending on where the compensating block was injected, so we chose to generate preimages of the public key CMAC instead. With a forged RSA public key, we were then able to sign arbitrary microcode patches without having to work around these compensating blocks.

In order to generate preimages of the public key CMAC, we simply generated candidate RSA public keys that collided with the expected CMAC value, and checked to see if they could be factored. After a few attempts we found a number that had prime factors that were easy to find (in the RSA algorithm these would be the equivalent of p and q which can then be used to generate the private key):

```
N = 0x151d07eae2f8151d07eae2f8151d07eaffb72072d
718ba6b0695e24f3aaa01e24f2cddb0d3224f2cddb0d322
4f2cddb0d3224f2cddb0d3224f2cddb0d3224f2cddb0d32
24f2cddb0d3224f2cddb0d3224f2cddb0d3224f2cddb0d3
224f2cddb0d3224f2cddb0d3224f2cddb0d3224f2cddb0d
3224f2cddb0d3224f2cddb0d3224f2cddb0d3224f2cddb0
d3224f2cddb0d3224f2cddb0d3224f2cddb0d3224f2cddb
0d3224f2cddb0d3224f2cddb0d3224f2cddb0d3224f2cdd
b0d3224f2cddb0d3224f2cddb0d3224f2cddb0d3224f2cd
db0d3224f2cddb0d3224f2cddb0d3224f2cddb0d3224f2c
ddb0d3224f2cddb0d3224f2cddb0d3224f2cddb0d3224f3
* 0x61

0x80000000000000000000000000000000ae4634b83805e
```

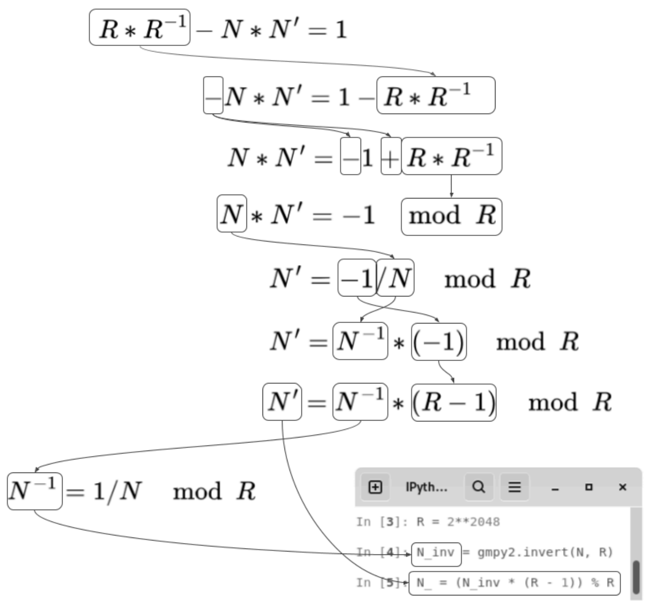
```
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000013
```

The sixteen byte value *ae4634b83805ea28d7ecac0053a6ab6c* is there simply to compensate and generate the collision.

The reason we were able to find a 2048-bit number that we could factor this easily is that, while an RSA public key is hard to factor, a random (odd) integer in the same size range is very likely to be the product of a few small primes with only one large prime. In this case, a factorization can easily be computed, for example, by dividing the number by small primes to see if they have a zero residue and then testing if the leftover is likely to be a prime. While RSA is usually computed using two large and distinct primes, the mathematics would hold for a product of multiple primes.

With this we could then just sign arbitrary microcode patches without having to inject compensating blocks on the microcode as long as the CPU accepted our public key (which consists of our generated modulus and the public exponent 65537, which was hardcoded for efficiency reasons). However, there was still one more thing we needed to do. Since the implementation uses Montgomery Modular Multiplication (see this video and this summary for an explanation of this algorithm) to simplify the signature verification algorithm, we needed to provide one more value that the signature verification scheme would check before trusting our public key.

compute, that R has to be coprime with our public key modulus (N), and also bigger than it. AMD chose R to be 2**2048 because modular operations modulo a power of 2 are easy to compute and the modulus size is 2048 bits (and since N is a product of two primes, it is guaranteed to be coprime with a power of two). To use Montgomery reduction, we need the constant called N' (or N_) that is defined by:


$\boxed{R * R^{-1}} - N * N' = 1$
$\boxed{-}N * N' = 1 - \boxed{R * R^{-1}}$
$N * N' = \boxed{-}1 \boxed{+} \boxed{R * R^{-1}}$
$\boxed{N} * N' = -1 \quad \boxed{\bmod R}$
$N' = \boxed{-1}\boxed{/N} \quad \bmod R$
$N' = \boxed{N^{-1}} * \boxed{(-1)} \quad \bmod R$
$\boxed{N'} = \boxed{N^{-1}} * \boxed{(R-1)} \quad \bmod R$
$\boxed{N^{-1}} = 1/N \quad \bmod R$
IPyth...

```
In [3]: R = 2**2048
In [4]: N_inv = gmpy2.invert(N, R)
In [5]: N_ = (N_inv * (R - 1)) % R
```

Fig. 8. Definition of the constant N' (or N_)

Putting it all together:

```
R = 2**2048
N_inv = gmpy2. invert (N, R)
N_ = (N_inv * (R - 1)) % R
bin((N_ * N) % R) = 0b111...111
```

The microcode verification routine verifies that it has the correct N_ by checking that the bottom 2048 bits of N * N_ are all ones (equivalent to -1); if this is not the case, it will return an error. With this value, multiplications can then be done using Montgomery modular

Vulnerability Mitigation

The fix released by AMD modifies the microcode validation routine to use a custom secure hash function. This is paired with an AMD Secure Processor update which ensures the patch validation routine is updated before the x86 cores can attempt to install a tampered microcode patch. We plan to provide additional details in the upcoming months on how we reverse engineered the microcode update process, which led to us identifying the validation algorithms, extracting the CMAC key, and discovering some file format details.

Peeking Inside the CPU

Before we move on to writing arbitrary microcode, we should first quickly discuss how AMD's microcode is implemented and executed. There is limited prior work investigating AMD's microcode instruction set and implementation details; namely the "The anatomy of a high-performance microprocessor" book on CPU microarchitecture from 1998 contains content on AMD's K6 RISC86 architecture, and Koppe, et. al published "Reverse Engineering x86 Processor Microcode" in 2017 which documents their reverse engineering efforts against the AMD K8 & K10 CPUs from the mid 2000s. To the best of our knowledge, the current AMD microcode design stems from the NexGen x86 CPUs from the mid 90's. These CPUs utilized an internal RISC-like architecture coined "RISC86" to implement x86 instructions. AMD ended up acquiring NexGen in 1996 and launched the AMD K6 CPU using NexGen's RISC86 technology. While many details have changed over the years (see this site with a collection of resources from old NexGen CPUs), the fundamental design of Zen microcode shares some similarities to the original NexGen design.

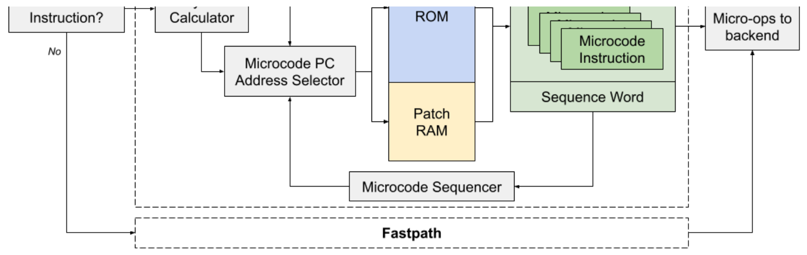


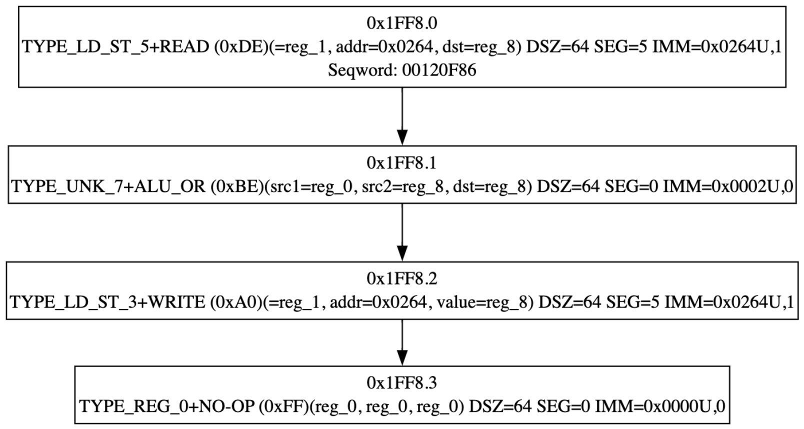
Fig. 9. Microcode engine design

When the CPU frontend decodes the next x86 instruction, it makes a decision on whether or not the instruction is microcoded or can take a “fast path” by directly emitting micro-ops. Many simple instructions are decoded directly into a small number of micro-ops which are passed on to the CPU backend (and cached in the micro-op cache so next time an instruction at that address has to be executed, the decoding step can be skipped). For more complex instructions and other system operations, control instead passes to the microcode engine which in turn can emit a series of micro-ops. Circuitry in the CPU determines the entry point in the microcode ROM for the current x86 instruction (e.g., `RDRAND => 0x0543`) and the microcode engine begins execution from there. The microcode ROM consists of fixed length 64-bit microcode instructions (micro-ops); these are bundled together into sets of four (quads) and terminated with a sequence word. The sequence word is a 32-bit command word which instructs the microcode engine what to do next. The microcode engine also utilizes branch delay slots for these sequence words, meaning that a specified branch won't occur until _after _execution of the next quad.

The specification, format, and semantics of the microcode instructions and sequence words are all undocumented. We spent a lot of time working on reverse engineering this information (both before and after gaining code execution), but much work remains and there’s a chance some bits may never be fully understood. Keep this in mind when reviewing our results and tools – there are likely pieces we

As a simple example, our current disassembly of the microcode instructions related to the Retbleed patch are listed below. This is one of the few microcode patches where AMD published any details on what the patch does:

> AMD "Zen 2" CPUs support a configuration bit in MSR C001_10E3 (DE_CFG2) which changes the behavior of the decode block when the processor attempts to predict a branch on a non-branch instruction (BTC-NOBR case). When bit 1 (SuppressBPOnNonBr) is set the branch prediction information on non-branch instructions is ignored and no speculation at the predicted target will be observed. Setting this bit mitigates the risk of potential information disclosure as a result of speculation in the BTC-NOBR case.
>
> **Some systems with recent microcode updates installed may already have this MSR bit set to 1.**



```
0x1FF8.0
TYPE_LD_ST_5+READ (0xDE)(=reg_1, addr=0x0264, dst=reg_8) DSZ=64 SEG=5 IMM=0x0264U,1
Seqword: 00120F86

0x1FF8.1
TYPE_UNK_7+ALU_OR (0xBE)(src1=reg_0, src2=reg_8, dst=reg_8) DSZ=64 SEG=0 IMM=0x0002U,0

0x1FF8.2
TYPE_LD_ST_3+WRITE (0xA0)(=reg_1, addr=0x0264, value=reg_8) DSZ=64 SEG=5 IMM=0x0264U,1

0x1FF8.3
TYPE_REG_0+NO-OP (0xFF)(reg_0, reg_0, reg_0) DSZ=64 SEG=0 IMM=0x0000U,0
```

Fig. 10. A disassembled view of the microcode patch released by AMD to mitigate the Retbleed vulnerability

For this microcode patch, we believe it is executed upon completion of the microcode update routine itself and thus does not hook a specific instruction or microcode ROM address. The first micro-op

The second micro-op does a bitwise OR of register t8 with 0x0002. The third instruction writes the updated value of register t8 back to memory segment 5 at offset 0x264. Finally, the last instruction is a no-op. In this example, we don't fully understand the sequence word. Put together, this sequence appears to mimic the description in AMD's security bulletin – it sets the `SuppressBPOnNonBr` bit in the `DE_CFG2` MSR (implemented in microcode memory) when loaded.

An unrelated but more complex sequence in the same microcode patch is shown below. From this, you can begin to see the extent and limitations of our reverse engineering progress as well as a hint of what's possible in microcode. Here we see the ROM address 0x111A being hooked – we currently don't know what was originally implemented at this address. The first basic block appears to perform a bitwise AND on a temporary register and conditionally branch based on the result. Unfortunately, we don't have access to the original ROM so it is difficult to understand what content this register is expected to contain. The conditional branch is also interesting since it jumps to another patch quad at 0x1FCD; otherwise, the remaining two instructions execute before falling through to the same second quad. The other conditional branch jumps to the ROM address 0x041E and we see a follow-on connection to 0x042A – this is based on our understanding of the original sequence word (0x120**42A**) and the delay slot branch mechanics.

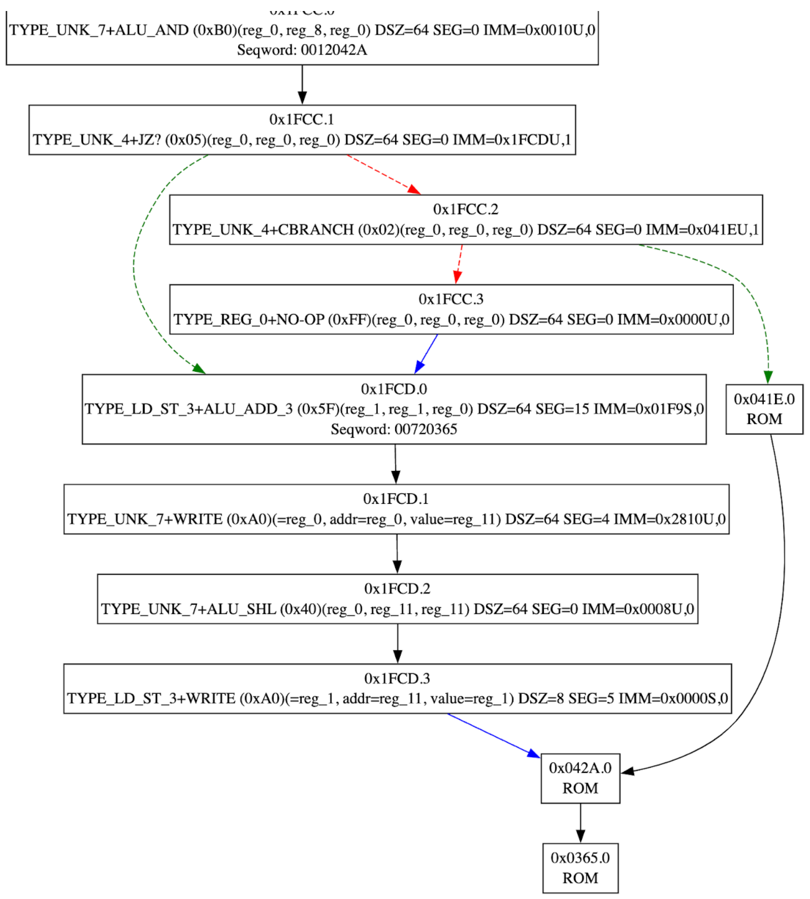

TYPE_UNK_7+ALU_AND (0xB0)(reg_0, reg_8, reg_0) DSZ=64 SEG=0 IMM=0x0010U,0
Seqword: 0012042A
0x1FCC.1
TYPE_UNK_4+JZ? (0x05)(reg_0, reg_0, reg_0) DSZ=64 SEG=0 IMM=0x1FCDU,1
0x1FCC.2
TYPE_UNK_4+CBRANCH (0x02)(reg_0, reg_0, reg_0) DSZ=64 SEG=0 IMM=0x041EU,1
0x1FCC.3
TYPE_REG_0+NO-OP (0xFF)(reg_0, reg_0, reg_0) DSZ=64 SEG=0 IMM=0x0000U,0
0x1FCD.0
TYPE_LD_ST_3+ALU_ADD_3 (0x5F)(reg_1, reg_1, reg_0) DSZ=64 SEG=15 IMM=0x01F9S,0
Seqword: 00720365
0x041E.0
ROM
0x1FCD.1
TYPE_UNK_7+WRITE (0xA0)(=reg_0, addr=reg_0, value=reg_11) DSZ=64 SEG=4 IMM=0x2810U,0
0x1FCD.2
TYPE_UNK_7+ALU_SHL (0x40)(reg_0, reg_11, reg_11) DSZ=64 SEG=0 IMM=0x0008U,0
0x1FCD.3
TYPE_LD_ST_3+WRITE (0xA0)(=reg_1, addr=reg_11, value=reg_1) DSZ=8 SEG=5 IMM=0x0000S,0
0x042A.0
ROM
0x0365.0
ROM

Fig. 11. A more complex example of a sequence in a microcode patch released by AMD

Zentool: Putting all the Pieces Together

Now that we have examined the vulnerability that enables arbitrary microcode patches to be installed on all (un-patched) Zen 1 through Zen 4 CPUs, let's discuss how you can use and expand our tools to author your own patches. We have been working on developing a collection of tools combined into a single project we're calling *zentool*. The long-term goal is to provide a suite of capabilities similar to binutils, but targeting AMD microcode instead of CPU machine code. You can find the project source code here along with documentation on how to use the tools.

limited amount of reverse engineered assembly, microcode patch signing, and microcode patch loading. We plan to also release details on how to decrypt and encrypt microcode patches in the future. A significant portion of the ongoing research is focused on building an accurate understanding of the AMD microcode instruction set – the current disassembly and assembly are not always accurate due to this challenge.

Let's walk through some brief examples of using the tool; for more in-depth information, please refer to the documentation in the *zentool* repository. Previously, we released a proof-of-concept microcode patch which hooked the RDRAND instruction and changed its behavior so that it always returns 4 (or sometimes returns 5 as one observant person noted). Here are the steps needed to reproduce this custom AMD microcode patch:

1. Clone the zentool repo and build the tools. Additionally, you need to construct a donor microcode patch. This is done by taking an official microcode patch for our current CPU and stripping out all of the original patch contents.

```
$ git clone https://github.com/google/secu
rity-research.git
$ cd security-research/pocs/cpus/entrysig
n/zentool
$ make all template.bin
```

2. Replace the donor patch contents with the necessary changes to replace RDRAND's implementation.

```
$ ./zentool -o rdrand.bin edit --match 0=@
rdrand --seq 0=0x100002 --insn q0i0="mov.q
s rax,rax,4" template.bin
```

An explanation of the required arguments:

- `--o rdrand.bin`: This instructs zentool to save the result into a new file.
- `edit`: This selects the `edit` subcommand which is used to modify the microcode patch contents.
- `--match 0=@rdrand`: This changes match register 0 to point to the entry point of RDRAND in the microcode ROM. The match registers direct the microcode sequencer when to execute instructions in patch RAM instead of ROM. We have identified a limited number of associations between instructions and ROM addresses, see `matchscan.sh` for details.
- `--seq 0=0x100002`: This sets the sequence word for the first two instruction quads to the value 7. This essentially instructs the microcode sequencer to return back to the next x86 instruction after the micro-ops in these quads have executed.
- `--insn q0i0="mov.qs rax,rax,4"`: This specifies the microcode instruction to place at quad 0, instruction slot 0. The `mov` operation will store the value 4 in the x86 register RAX. The `q` suffix indicates that the value is a 64-bit QWORD and the `s` suffix indicates that RFLAGS should be set based on the operation.

(Note: This example has been simplified to only set RAX and does not set the carry flag as expected by a successful RDRAND execution.)

3. This command takes the modified microcode patch file and signs it using our fake RSA key as described in the previous sections. At this point, CPUs without the EntrySign fix can not discern this patch from any authentic AMD microcode patch.

```
$ ./zentool resign rdrand.bin
```

4. This command simply loads the RDRAND patch on a specific CPU using the standard WRMSR technique:

```
$ sudo ./zentool load --cpu=2 rdrand.bin
```

Writing and running a simple program that executes the RDRAND instruction will demonstrate that the instruction always returns 4.

```
#include <stdbool.h>
#include <stdio.h>
int main(void)
{
  unsigned long r;
  bool ok = false;
  asm volatile(
    "rdrand %0\n\t"
    "setc %1"
    : "=a"(r), "+r"(ok));
  printf("ok? %u, val %#lx\n", ok, r);
}
```

You may want to use the `taskset` Linux program to force execution on the patched core as well as the `isolcpus` kernel command line option to prevent other jobs from running on the patched CPU.

```
$ gcc rdrand.c -o rdrand

# Normal operation
$ taskset -c 0 ./rdrand
ok? 1, val 0xda8cb8920d9e381c

# Patched operation
$ taskset -c 2 ./rdrand
ok? 0, val 0x4
```

Future Work

We hope you now have a better understanding of AMD's microcode and how flaws in their signature verification scheme led to a breakdown in trust for microcode patches. Luckily, the security impact was limited by the fact that attackers must first obtain host ring 0 access in order to attempt to install a microcode patch and that these patches do not persist through a power cycle. Confidential computing using SEV-SNP, DRTM using SKINIT, and supply chain modification are some of the situations where the threat model permits an attacker to subvert microcode patches. We have examined AMD's patch and believe it remediates the signature verification vulnerability.

Beyond the security impact, we are looking forward to seeing what the security (and non-security!) research community can produce with this new capability. Previous research on Intel microcode has demonstrated the ability to craft new instructions to implement security features similar to ARM's pointer authentication codes, accessing internal CPU buffers, tracing microcode, and more. Today we have published our main tool suite, *zentool*, which enables researchers to begin examining microcode patches and creating their own; bug reports and contributions are welcome. Additional reverse engineering and testing is required to begin to fully understand the work progresses.

BACK TO OVERVIEW