CPU covert channel accessible from JavaScript

James Sutherland, Natalie Coull and Allan MacLeod

University of Abertay, Dundee, UK

1307716@live.abertay.ac.uk, N.Coull@abertay.ac.uk, A.MacLeod@abertay.ac.uk

Abstract: The combination of dedicated cryptographic instructions on modern processors and the revelation that government agencies had devoted substantial resources to a variety of hardware backdoors raised interesting questions about possible techniques to compromise or ensure their integrity. This paper describes how modifications to a processor implementing the x86/x64 instruction set can be used to construct a backdoor through which an unprivileged user merely viewing a modified webpage can expose AES encryption keys to an attacker, without disrupting normal operation of the processor or being detectable by normal investigative methods. As a proof of concept, this was used to capture an AES encryption key for later covert retrieval: first, encrypting a file using standard tools, then viewing a web page modified for the purpose. With no visible anomalies, the encryption key used had been captured by the web server and emailed to the demonstrating attacker. Potential countermeasure techniques are also examined, including those which could be implemented in software alone and verified to be operating effectively using typical test equipment.

Keywords: Covert channel, backdoor, processor, JavaScript, data exfiltration

1 Introduction

Hardware backdoors have been a popular topic recently, perhaps encouraged by the Snowden revelations and speculation about possible NSA activities in the field. In particular, Daniel J Bernstein asked publicly [1] “Thought experiment in malicious chip design: What would be the easiest way for an Intel CPU to leak AESKEYGENASSIST inputs to an attacker?”

This is an instruction in Intel’s AES-NI instruction set extension, used to access an on-chip AES encryption engine; this particular instruction generates the expanded AES key schedule given the key 128 bits at a time, giving it direct access to the key itself (the other AES-NI instructions operate on the expanded key schedule instead, complicating key extraction slightly.)

Loïc Duflot showed in 2008 [2] that a processor can be modified to incorporate a backdoor for later use by unprivileged applications running on the affected system. This enables privilege escalation from user to kernel mode through execution of a modified x86 instruction with specific values in each of four registers. Biham, Carmeli and Shamir found [3] that incorrect calculation of a single multiplication when performing RSA decryption operations can, in certain circumstances, function as a covert channel disclosing the target’s private key to the attacker.

The essence of a backdoor is that an attacker has control over the target device initially, and wishes to be able to regain some control or access in future. In the case of a CPU design-level backdoor, this would typically entail having compromised the original design or manufacture of the processor, or covertly modified or replaced it with a modified device of your own. Later, the attacker would have limited access, which could be used to access the backdoor and obtain additional access or information — in this case, recover cryptographic keys previously used on the processor and covertly retained.

Given full control over the software executed, even in an unprivileged mode, the covert channel would be quite trivial: any otherwise-invalid operation code, or atypical usage of a valid one, would suffice. This, however, requires the ability to execute custom software directly on the target system, which entails a substantial level of access; making the backdoor instruction available through sandboxed interpreted code such as JavaScript imposes much tighter constraints. The NSA are known to have used the insertion of JavaScript [4] for similar purposes already.

The proof of concept backdoor presented here consists of a processor modification (which could have been embedded in the design or manufacture stage, or incorporated in a replacement counterfeit processor, by a resourceful attacker such as the NSA), simulated in software in the Bochs system simulator, and a trivial webpage with embedded JavaScript to access this backdoor.

2 Implementation

To render the backdoor accessible from JavaScript, yet extremely difficult to detect without prior knowledge of the specific implementation, a standard mathematical operation is required — one which the JavaScript implementation will be very likely to map directly to the equivalent machine instruction. To implement the backdoor, this instruction will be modified to respond differently to a particular set of operands. Duflot's demonstration targeted a specific x86 instruction, SALC ('set register AL's contents according to the Carry processor flag', opcode 0xD6) in conjunction with specific register contents, which is of course not directly accessible from JavaScript or indeed any other high-level language.

All JavaScript numbers are defined to be IEEE-754 64 bit double-precision values — there is no integer type — and the use of a dyadic non-commutative instruction such as FDIV ensures the two operands will be presented to the CPU in the same order in which they are encoded in JavaScript. A specific ordered pairing of 64 bit floating point values represents almost a 128 bit shared secret for activation of the backdoor: ignoring the reserved and special bit sequences used in floating point representations of NaN and infinities, an adversary would have to determine which of almost 2^{128} possible value-pairs is being used.

Ethan Heilman's response [5] to Bernstein's enquiry proposed the use of 64-bit integer multiplication as the instruction for accessing the backdoor, but being a commutative instruction, this would be slightly harder to recognise in hardware — the implementation would have to respond to either order of the two component values — and would also not map cleanly to JavaScript, which has no integer type and no standardised support for handling 64 bit integers, since the IEEE-754 format it uses has only 53 bits of precision (the mantissa).

The proof of concept discussed here incorporates these changes in the Bochs open-source system simulator. Unlike most virtualisation products, which rely heavily on executing the simulated system's instructions directly on the host processor wherever possible, Bochs simulates every instruction in software, bringing a substantial performance overhead but giving the flexibility needed to modify the simulated processor's instruction set in this way.

For these purposes, the attack captures just the 128-bit operand used each time the AESKEYGENASSIST instruction is invoked to generate the round 1 part of a key schedule: with 128-bit AES, this will be the entire key. (The 192 and 256 bit variants split the key between the first and second invocations of this instruction, so capturing the second operand as well would cover these variants.) These are stored in a hidden buffer within the processor for later retrieval.

The modified FDIV implementation then returns the data from this hidden buffer, 32 bits at a time, as the result of dividing a particular random pair of floating-point numbers. For this experiment, the arbitrary value $1.91207592522993 \times 10^{306}$ was used for both halves of the pair. Where an unmodified processor would return the correct result of this division — in this case, 1 — the modified one instead returns the next 32 bit portion of the concealed key each time, converted to an equivalent floating point value. Repeating the calculation four times and concatenating the results yields the original key.

3 Results

As expected, an unmodified Windows XP system image booted and functioned correctly on the simulated system. After encrypting a file using the Win32 port of OpenSSL on the command line, a JavaScript script running under the included copy of Internet Explorer retrieved the key and was able to pass that data back to the server it was hosted on using a simple HTTP request — modifying the URL by which a GIF file containing a single transparent pixel was retrieved — with no user-visible effects. (Passing values back to the server in this fashion is routinely used for legitimate purposes, such as Google's PageSpeed 'beacon' used for profiling the performance of web pages, and insertion of random numbers in URLs is commonly used as a cache-busting mechanism for banner ads, so this would be very unlikely to attract the attention of any intruder detection system.)

Once implemented, loading a web page containing an IMG tag and the fragment of JavaScript shown in figure 1 resulted in the server receiving the HTTP request shown in figure 2 — the four hexadecimal numbers are the AES key used, split into four 32 bit integers.

```
// Perform the rigged FDIV four times:
var a=1.91207592522993E+306/1.91207592522993E+306;
var b=1.91207592522993E+306/1.91207592522993E+306;
var c=1.91207592522993E+306/1.91207592522993E+306;
var d=1.91207592522993E+306/1.91207592522993E+306;
if (a!==1) {  // Result will be 1 on untampered CPU
  var k=a.toString(16)
   +','+b.toString(16)
   +','+c.toString(16)
   +','+d.toString(16);  // Concatenate the 4 values

  // Now put that key in the URL of the first image:
  document.images[0].src='?n='+k;
}
```

Figure 1: Javascript exploit activation code

```
GET /floatback.php?n=efbeadde,77665544,bbaa9988,ffeeddcc HTTP/1.1
```

Figure 2: Resulting HTTP request to server

4 Discussion

The instructions targeted in this proof of concept are complex enough to make verification of their behaviour extremely difficult — indeed, Intel's implementation of FDIV on the original Pentium processors contained a substantially less subtle flaw, such that approximately one in 2^{33} divisions yielded incorrect results, as opposed to one in 2^{128} in this version, but still went undetected until after a substantial number of affected processors had been sold and used.

The two instructions concerned are also very likely to be heavily reliant on microcode (the flawed Pentium FDIV implementation relied on a 2,048 entry lookup table which was incorrectly loaded during manufacture), which raises the possibility that such a backdoor could be implemented in a malicious microcode update, rather than by modifications to the CPU itself.

A hardware implementation would be relatively trivial — a bitwise comparison of floating point multiplier operands, a low bit-rate path from the AES unit to the floating point unit (probably using the shift register in the existing processor test facilities)

5 Mitigation and detection

One common mitigation strategy against covert channels for data leakage is to separate applications in multiple virtual machines, as used in the Qubes operating system [6]. Sensitive cryptographic operations could be confined to one virtual machine, with no network access, while other activities take place in another, isolated from the cryptographic keys. Because the covert channel is unprivileged and unknown to the virtualisation platform, the stored key remains in place for the modified FDIV instruction to return later, regardless of context switches or other operations in the meantime. If the hypervisor had some mechanism to clear *all* processor state, including the hidden storage used for the covert channel, this would become an effective countermeasure to this attack.

Waksman and Sethumadhavan [7] propose design-level countermeasures to CPU backdoors, some of which would be effective at defeating this attack: in particular, they used frequent power-cycling of individual portions of the processor to defeat covert data retention. Without power, data cannot be retained for more than a brief time in volatile memory. They propose additional measures to ensure no non-volatile memory can be present: repeated use of the untrusted operation, AES key schedule expansion in this case, would trigger repeated writes to the non-volatile memory, rapidly exceeding the lifespan of any current flash-type storage.

As simulated in their research, however, this requires significant design changes to the processor; moreover, experiments [8] show that a momentary interruption of power is not sufficient to clear data, with 50% of bits remaining after 44ms without power in typical six-transistor static RAM (SRAM) on a 65nm device at $-40^{\circ}C$; at a more typical $25^{\circ}C$, however, 90% of bits are lost within 1ms. (Retention times are shorter for dynamic RAM — DRAM — than for SRAM, so SRAM remanence times can be used as an upper bound.)

A variation on this approach could be effective, however: putting a standard x86-architecture system in ACPI sleep mode S2 or higher suspends power to the processor until a wakeup signal is received. This could easily be verified in situ by simply checking the voltage: when in this sleep mode, power will no longer be present on the processor's voltage regulators on the system board, and all data within the processor will have been lost within a small fraction of a second. Having established that entering ACPI S2 or higher genuinely interrupts the processor's power while in effect, an interruption of 100 ms after any sensitive cryptographic operation would be sufficient to defeat covert data retention within the CPU in future, with minimal impact on system usage — 100 ms is a comparable period to a single operating system timeslice.

A more sophisticated implementation of this backdoor could persist the target data in non-volatile storage such as the system's BIOS chip, or more durable volatile memory such as the SMRAM (system management memory, a reserved section of main system memory protected from access by anything other than the CPU operating in System Management Mode) but this would be more difficult and significantly more observable: in particular, component-level monitoring of the system board could detect such write activity taking place each time an AES key schedule is generated, revealing the presence of such a compromise.

The specific covert channel of modified FDIV functionality accessed from JavaScript could be detected when used — a suitably modified JavaScript interpreter could verify the results of each division operation by multiplying the result by the divisor as a consistency check, with modest overhead; indeed, this is essentially the mechanism which exposed the original Pentium 'FDIV bug'

in 1994 [9]. That would only address JavaScript, and then only when in that modified browser, however.

Using AES-NI's hardware implementation of AES is normally preferable both for performance reasons and security advantages such as resistance to timing attack, to which software implementing AES appears particularly vulnerable with Bernstein demonstrating [10] full remote key recovery, although timing-invariant software implementations have now been shown to be possible [11]. Replacing the use of AESKEYGENASSIST with code to perform the key schedule expansion stage in software while still using AES-NI hardware for encryption and decryption should preserve most of these advantages while defeating any backdoor of this type.

Detection of a hardware implementation of this backdoor would appear extremely difficult. A transistor-level reverse engineering effort of the AES and floating-point division portions of the processor would be able to identify the presence of such modifications, by breaking down each layer of the chip in turn with acid or mechanical erosion then using scanning electron microscopy to map them. Since this is both an extremely expensive process and a destructive one, testing even a sample of processors would be impractical. The precautions described earlier, however — verifying that entering ACPI sleep mode 2 and higher genuinely interrupts CPU power, using memory and other bus analysers to verify that no unexpected accesses take place as a result of use of cryptographic instructions, then causing 100ms power interruptions to the CPU after any sensitive cryptographic operation and before less trusted operations. (Excluding the existence of write operations would not be sufficient: some hardware components are read-sensitive, so a sequence of read operations on particular addresses could also be a mechanism for storage.)

6 Future Work

Capturing cryptographic keys from algorithms other than AES would be an interesting extension, though more difficult since it would require recognising particular sequences of machine instructions; obfuscation of algorithms to defeat such recognition would also be interesting.

With integration of memory controllers on the processor die, it would be possible to implement a similar covert channel without the need for the attacker to have any instructions execute at all: in theory, the memory controller could respond to specific 'magic' values being written to memory by a device using DMA (such as a network card delivering an incoming packet) by substituting the data being exfiltrated. For packets such as an ICMP echo request ('ping') where the inbound packet is supposed to be returned unmodified, this would result in the attacker receiving the data directly. The practicality of this approach remains uncertain however.

The proposed countermeasure of briefly putting the system in an ACPI sleep mode in which the processor no longer receives power merits further investigation. It is normal to enter and return from this sleep mode in under two seconds (a target set by Microsoft for hardware approval of retail systems), including powering down and reactivating associated devices. This does not seem onerous as an occasional sanitisation step to be performed on high-security systems after any sensitive operation — but if optimisation can reduce this by between one and two orders of magnitude, to be comparable with a scheduler time-slice, it could be routinely and automatically performed by the hypervisor in a system such as Qubes whenever a high security virtual machine handling sensitive data is releasing the processor for use by others.

7 Conclusion

A CPU backdoor is shown in simulation to allow covert retention and subsequent exfiltration of AES keys, with no apparent impact on the system functionality and an extremely low probability of the backdoor being discovered.

Putting the system into ACPI sleep state S2 or higher for as little as a few milliseconds should be an effective countermeasure to this attack, though this has not yet been investigated in practice. Detecting the presence of such a backdoor in a processor, or proving the absence thereof, would appear extremely difficult without knowledge of the specific numbers in use as a trigger.

References

[1] Daniel Bernstein (Twitter: hashbreaker). https://twitter.com/hashbreaker/status/378258465291915264.

[2] Loïc Duflot. CPU bugs, CPU backdoors and consequences on security. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 580–599. Springer, 2008.

[3] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 221–240. Springer, 2008.

[4] Bruce Schneier. Attacking Tor: How the NSA targets users' online anonymity. [Online; accessed 06-April-2014].

[5] Ethan Heilman. https://twitter.com/Ethan_Heilman/status/378282494992187392.

[6] J Rutkowska and R Wojtczuk. Qubes OS architecture, 2010.

[7] A. Waksman and S. Sethumadhavan. Silencing hardware backdoors. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 49–63, May 2011.

[8] C. Cakir, M. Bhargava, and Ken Mai. 6T SRAM and 3T DRAM data retention and remanence characterization in 65nm bulk CMOS. In *Custom Integrated Circuits Conference (CICC), 2012 IEEE*, pages 1–4, Sept 2012.

[9] Vaughan Pratt. Anatomy of the Pentium bug. In *In TAPSOFT'95: Theory and Practice of Software Development*, pages 97–107. Springer Verlag, 1995.

[10] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[11] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. IACR Cryptology ePrint Archive, report 2009/129, 2009.