Exploring the viability of power management attacks on Intel Core processors

Dissertation for M.Sci. in Mathematics and Computer Science

Navjivan Pal (Student No. 1529469)
Supervised by Dr.-Ing. David Oswald

University of Birmingham

School of Computer Science

8 April 2019

Abstract

Almost all of today's computers, including desktops, laptops, and smartphones, feature the ability for the operating system to self-regulate the frequency and voltage at which their processor cores operate in order to manage power consumption. Recent research has demonstrated that these extremely prevalent power management features can be exploited with no more than a malicious kernel driver, though these CLKSCREW attacks, as they are called, have so far only been demonstrated in a single device class, namely ARM smartphones employing ARM TrustZone. We attempt to widen the scope of platforms to which such power management attacks are seen to apply by carrying over the principles of this prior research to commodity desktop processors belonging to the Intel Core family. We provide a proof of concept that computational faults can be injected successfully on this platform by judiciously setting power management parameters via the use of no more than a kernel module, and detail the processes involved in getting to this stage.

All source code developed in relation to this project can be found at:
`https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2018/nxp569`

A copy of the contents of this repository is included alongside this report in a ZIP file.
Please see the `README.txt` file included within the ZIP file for details.

Contents

1 Introduction and background

1.1 Role of power management in modern CPUs

Power management features in electronic devices are heavily desired and expected to be present in the devices of today's end-users for numerous reasons, including, but not limited to:

- reducing energy consumption, thereby also reducing electricity costs involved in operating a device;
- prolonging battery life, particularly in embedded and portable devices such as smartphones; and
- reducing the temperature of a device, thereby also reducing the need for extensive cooling solutions such as loud fans to mitigate overheating and user discomfort.

Power management is achieved in modern computers through various means, including:

- efficient application multi-tasking principles (foreground and background process management by the operating system);
- traditional power-saving measures, such as turning off the device's display or putting it into standby after a given length of time; and
- dynamic voltage and frequency scaling.

1.2 Dynamic voltage and frequency scaling (DVFS)

Dynamic voltage and frequency scaling (DVFS) is a feature of almost all contemporary devices. It allows the following aspects of a device's CPU cores to be adjusted via software (as opposed to needing to alter such parameters directly via hardware):

- the voltage supplied to them, a.k.a. core voltage; and
- their clock speed, a.k.a. frequency.

These parameters can each be:

- decreased in order to decrease energy consumption; or
- increased in order to increase computational power and speed.

The voltage and frequency of a device's CPU cores can thus be dynamically adjusted via software depending on the current computational demand in order to balance the trade-off between energy efficiency and computational power; a device can afford itself the ability to perform complex, labour-intensive tasks when necessary, whilst using as little power as possible at other times.

In common parlance, the terms *overclocking* and *underclocking* are used to refer to increasing and decreasing the processor's frequency, respectively. Similarly, *overvolting* and *undervolting* refer to increasing and decreasing the core voltage, respectively. A given frequency–voltage pair is called an *operating performance point* (OPP) or *performance state* (P-state).

1.3 Security issues due to DVFS

DVFS poses security issues due to the nature of the design of CPU microarchitectures. Microprocessors comprise many millions or billions of transistors, which serve the role of storing and propagating information within the CPU as and when required. One such type of transistor is the flip-flop, which has an input line, an output line, and a gate. When the gate signal is high (i.e. it has a high voltage relative to ground), the signal that is currently being input to the flip-flop is allowed to propagate through to the output.

A common scenario in CPUs is to have two flip-flops in series (the output of one being connected to the input of the other), with the gate of each flip-flop being tied to the processor's clock signal. Thus, with each pulse of the clock signal, each flip-flop sends its received input along the chain to the next one. This presents an issue if the amount of time that it takes for a signal to propagate from one flip-flop to the next is *longer* than the length of the clock cycle, as then, if the second flip-flop is supposed to propagate a signal from the first, it will not receive this signal in time for it to be propagated. If the signal to be sent was opposite from the previous signal, this manifests as a bitflip from the intended signal. For example, if the previous signal was a binary 0 (low voltage), and the next signal is a binary 1 (high voltage), but this is not received in good time, then a binary 0 is re-sent instead; the intended 1 becomes a 0.

This behaviour can be induced simply by tweaking the DVFS parameters such that the duration of the clock cycle is sufficiently longer than the amount of time it takes for a signal to propagate from one flip-flop to the next in a series of flip-flops.

1.4 Contribution and outline

We expand upon the work done in developing the CLKscrew exploit [1], applying the principles detailed there to attempt to demonstrate that the power management design of the Intel Core series of processors is subject to the same vulnerabilities. Firstly, we explore the extent to which software-accessible interfaces for DVFS exist on the platform in §3.1. Secondly, we use these interfaces to determine which OPPs cause system instability in §3.2. Finally, in §3.3, we devise a kernel module which temporarily puts the system into an unstable state whilst computing a SHA-1 hash in the hopes that an incorrect hash is computed; such an observation would serve to demonstrate that a power management attack is at least achievable and perhaps exploitable on this platform, though it does not constitute a proper attack in and of itself.

2 Related work

In this section, we cover existing literature relevant to the subject matter of this report. This report is, in large part, a direct response to the research pioneered by Adrian Tang et al. in his paper on CLKscrew [1] published in August 2017. There does not appear to be any directly relevant literature published prior to this, though similar attack vectors have been explored in the past, such as the Rowhammer exploit [2] first published in June 2014. Here, we cover the existing work in [1] and a related paper concerning the Blacklist Core technology [3] designed to defend against CLKscrew-style attacks. We also touch on the existence of Intel's trusted execution environment, dubbed Software Guard Extensions (SGX), and how the power management attacks may be a suitable attack vector for SGX, as also mentioned in [4]. In fact, the work detailed in this report could lay the foundation for developing a power management attack against Intel SGX, as discussed in §4.2.

2.1 CLKscrew

The CLKscrew paper [1] details a newly discovered (circa 2017) family of attacks dubbed CLKscrew that exploit the security vulnerabilities of DVFS discussed in §1.3. The authors detail how the secure execution environment, namely ARM TrustZone, of a Nexus 6 smartphone can be compromised via this attack vector, describing two successful attacks on the platform.

In particular, the goal of the first attack is to extract an AES key. This is done by inducing a precisely timed computational fault during decryption of a ciphertext to yield an incorrect plaintext. The fault is induced with the aid of a malicious kernel driver that alters DVFS parameters at the appropriate times once it detects that the AES decryption program is invoked within ARM TrustZone. This malicious code must be executed in kernel mode since the instructions that allow DVFS parameters to be altered are privileged. The incorrect plaintext can then be analysed alongside the correct plaintext to yield a relatively small set of key hypotheses via a technique known as differential fault analysis [5]. We ultimately seek to replicate this attack on the Intel Core platform in order to substantiate the theory that this attack vector is viable across many, if not all, modern CPUs that support DVFS via software, as discussed in [1, §6.1].

We note that the attacks demonstrated on the Nexus 6 rely on the fact that DVFS is possible on a per-core basis. That is, each CPU core of the device has its own DVFS parameters, allowing the malicious kernel driver to run on one core (the attacking core) whilst the target process (i.e. the AES decryption program being executed in ARM TrustZone) is pinned to another core (the victim core). This allows the malicious code to only cause instability on the victim core, as opposed to doing so on both the victim core and attacking core, which could result in faults occurring in the malicious code itself, which is undesirable from the attacker's perspective. In our research, we explore the possibility of reliably inducing faults when no such per-core DVFS regulation is possible, i.e. when all CPU cores are forced to use the same frequency and voltage as each other.

2.2 Blacklist Core

It is only natural to want to design mitigations against attack vectors, and CLKscrew is no exception. The Blacklist Core technology described in [3] is one such mitigation design, involving additional constructs within the CPU and a machine-learning algorithm. During normal operation of the device, the algorithm will attempt to determine those OPPs in which computational faults can occur and blacklist them so that in the event of such an OPP being requested in the future, the request can be denied. In theory, this would certainly suffice to completely prevent CLKscrew-style attacks, but this is dependent on the accuracy of the blacklisting algorithm. If the algorithm is lenient, some OPPs which could cause faults may be permitted; if it is conservative, many OPPs that do not cause faults but which could be of valuable use to the end-user will be denied.

It is already standard practice that hardware vendors conservatively stipulate their own recommended OPPs for devices, as can be seen in the DVFS drivers that they provide for devices using their hardware. The efficacy and purpose of a design such as that of Blacklist Core are therefore questionable: rather than implement a complex machine-learning algorithm in CPUs, perhaps it is better to simply hard-code these vendor-stipulated OPPs into the relevant hardware so that requests to set DVFS parameters beyond these conservative limits are always denied.

2.3 The Security of Intel Software Guard Extensions (SGX)

The possibility of Intel SGX being vulnerable to power management attacks is discussed in [4]. In particular, the author writes:

> "We remark that the CLKSCREW attack was not demonstrated specifically against SGX. However, as the researchers argue, the attack is relevant for all hardware that enables energy management (which is essentially all modern platforms). We note that AWS does provide processor state control for EC2 instances as required for the CLKSCREW attack, but stress that the attack has not been demonstrated on this specific platform. Thus, the question of the attack's relevance to SGX in real cloud environments is still open."

Whether SGX is indeed vulnerable to the same attacks demonstrated against ARM TrustZone detailed in [1] is still up for debate, hence our desire to demonstrate them ourselves. We do not actually do so in this report, but the desire to do so in future is discussed in §4.2.

3 Methodology

In this section, we discuss the interfaces available to us that allow us to set DVFS parameters. We determine those OPPs which render the system unstable, and then use the acquired information in §3.3 to develop a kernel module which alters the DVFS parameters at appropriate times in order to induce a computational fault which we can observe. Throughout this report, the testbench we use is an Intel Core i5-4590 desktop computer running Arch Linux with Linux kernel version 5.0.4.

3.1 Available DVFS interfaces

3.1.1 Frequency scaling

CPU performance scaling in the Linux kernel

The Linux kernel natively supports frequency scaling as part of its `CPUFreq` subsystem, documented in [6]. Precisely one scaling driver is registered with the subsystem during boot time, which implements at least one scaling governor: an algorithm that determines which OPP each CPU core should use at any given moment. Precisely one governor is in use at any given time, and this can be changed whilst the system is running.

Standard governors include:

- `performance` — enforces use of the most powerful OPP, at the expense of having poor energy efficiency;
- `powersave` — enforces use of the least powerful OPP, at the expense of having limited computational power;
- `schedutil` — uses data available from the CPU scheduler to make an educated guess as to which OPP is most suitable for the current workload; and
- `userspace` — allows userspace utilities to request use of a particular OPP via SysFS.

The standard governors listed above are implemented by the `acpi-cpufreq` driver, which can be used by the majority of modern CPUs, including those in the Intel Core series. However, processers in the Intel Core series from the 2nd generation (Sandy Bridge) and later notably have a much larger, more granular set of P-states than is typical, with the CPU itself managing which specific P-state to use. This is in contrast to the decision being made by the operating system, which then instructs the CPU to use the chosen P-state. This self-selection feature is known as HWP (hardware-managed P-states or hardware-coordinated P-states) [7, Vol. 3, §14.3.2.3]. In order to take advantage of HWP, the `intel_pstate` driver [8] was devised, which only implements the following scaling governors:

- `performance` — behaves as detailed above; and
- `powersave` — despite having the same name as the `powersave` governor of `acpi-cpufreq`, this governor instead behaves like the `schedutil` governor of `acpi-cpufreq`, choosing a P-state most suitable for the current workload based on information provided by the CPU's own feedback registers.

Observe that there is no analogue to the `userspace` governor; neither of the two governors offered by `intel_pstate` allows userspace processes to directly choose which P-state should be used. In order to overcome this barrier, we can choose not to use the `intel_pstate` driver by specifying `intel_pstate=disable` on the kernel command line (i.e. before boot), at which point the kernel will default to using `acpi-cpufreq` instead.

Setting the CPU frequency

The `CPUFreq` subsystem exposes an interface in SysFS that allows us to set the current P-state as desired. A front-end utility called `cpupower` exists to more easily use this SysFS interface, and is available in the Arch Linux repositories as detailed in [9]. We use `cpupower` in §3.2.

When using the `acpi-cpufreq` driver, we find that the range of possible frequencies is quite limited. SysFS tells us via a file named `scaling_available_frequencies` that the following options are available:

- 800 MHz
- 1000 MHz
- 1200 MHz
- 1300 MHz
- 1500 MHz
- 1700 MHz
- 1900 MHz
- 2000 MHz
- 2200 MHz
- 2400 MHz
- 2600 MHz
- 2800 MHz
- 2900 MHz
- 3100 MHz
- 3300 MHz
- 3301 MHz

The last of these options, 3301 MHz, is actually the driver's way of exposing the ability to use Intel Turbo Boost [8]. Our CPU's base frequency is 3300 MHz with a maximum Turbo frequency of 3700 MHz [10]. Requesting that the CPU frequency is set to 3301 MHz or higher simply causes the CPU to exclusively use Turbo P-states, meaning that CPU core will maintain frequencies between 3400 MHz and 3700 MHz for as long as physically possible, with each core's individual frequency varying depending on the workload. For the sake of our analysis, we treat this 3301 MHz option as representing the maximum Turbo frequency of 3700 MHz.

3.1.2 Voltage scaling

Intel does not seem to provide any documentation for the voltage scaling interface in their Intel Core processors, though a closed-source Windows utility called ThrottleStop [11] exists to control DVFS parameters. This utility has been reverse-engineered [12], revealing that a model-specific register (MSR) exists in this line of processors to read and write settings related to voltage scaling. Specifically, the MSR with address 0x150 acts as an interface for voltage scaling, although this is also not officially documented by Intel in [7, Vol. 4, §2.13] as would be expected.

Using this interface, it is possible to adjust the voltage offset from the base voltage in units of $\frac{1}{1024}$ volts. The MSR uses an 11-bit signed integer to describe the offset from the base voltage in these units, thus allowing a maximum voltage offset of ± 1 volts. A front-end utility called `undervolt` [13] has been developed in Python to more easily use the interface described in [12]. We use `undervolt` in §3.2. We use the MSR interface directly in §3.3.

3.2 Determining the stability of operating performance points

In this section, we determine which OPPs are stable and which OPPs are unstable, i.e. those in which the system operates normally and those which cause the system to crash, respectively. Such a crash typically manifests as a kernel panic. In particular, we would like to know which specific OPPs lie on the boundary between the stable realm and the unstable realm. Henceforth, such an OPP is referred to as a *critical point*, and the voltage offset associated with a critical point is called the *critical voltage* of the frequency associated with that same critical point. The following basic process lays out how we can determine these critical points:

(1) Choose a frequency that has not yet been chosen from the set of possible frequencies, and set that as the current CPU frequency. If no previously unchosen frequency exists, we are done and have determined all the critical points.

(2) Perform some fixed computational task, e.g. compute the SHA-1 hash of some predefined data. If the system crashes, record the current OPP as a critical point and go to step (1).

(3) Decrease the CPU voltage by the smallest possible amount (i.e. $\frac{1}{1024}$ volts), then go to step (2).

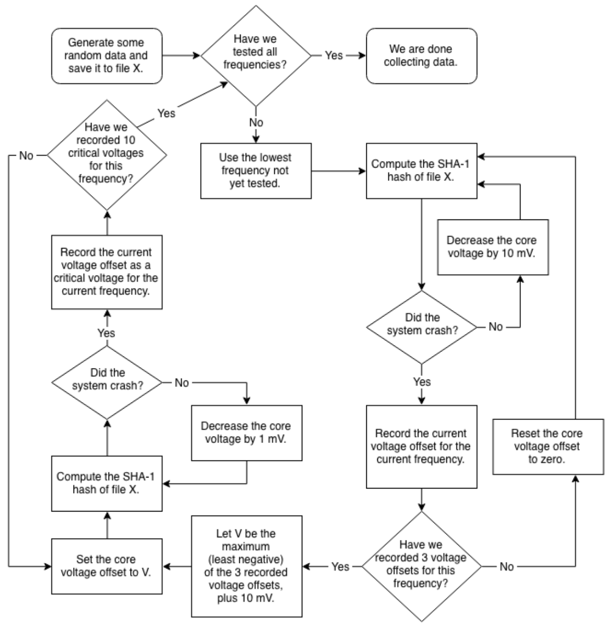
In practice, we expand upon this process in order to collect a more suitable dataset. The process given above is repeated numerous times for each possible frequency in order to obtain a reasonable sample size from which to determine an average voltage offset which renders the system unstable for that frequency. Ideally, this data collection would be automated, with the testbench running through the process detailed above on boot, recording the tested OPPs to disk, and rebooting after a system crash to repeat the process for different frequencies as required. In practice, this is not possible, for the following reasons:

- any data that ought to be written to disk may not actually be flushed from memory to disk before the system encounters a kernel panic, resulting in loss of these data; and
- the system does not always reboot after encountering a kernel panic, contrary to the operating system's intended function of rebooting 30 seconds after a kernel panic. In such situations, the machine must be reset manually.

Thus, to save time in collecting this data, we narrow down the range in which the critical voltage lies for each frequency by only testing certain voltage offsets. We then take the time to more precisely determine the critical voltage for each frequency by testing all voltage offsets within this narrower range. We do this with the aid of the `cpupower` and `undervolt` utilities discuss in §3.1.1 and §3.1.2, respectively.

The `undervolt` utility expects voltage offsets to be expressed in millivolts (mV, units of $\frac{1}{1000}$ volts) rather than the MSR interface's expected units of $\frac{1}{1024}$ volts. It then rounds the value given in millivolts to the nearest multiple of $\frac{1}{1024}$ volts and writes the appropriate value to the MSR. As such, the "smallest possible amount" mentioned in step (3) of the process above is taken as 1 mV rather than

$\frac{1}{1024}$ volts. The process we end up using is described by the flowchart in Figure 3.1. Since we cannot fully automate this process as discussed earlier, we collect the data by hand by running through this process with the aid of a shell script, `undervolt-test.sh`, the details of which are given in §3.4.1.



Generate some random data and save it to file X.
Have we tested all frequencies?
Yes
We are done collecting data.
No
Use the lowest frequency not yet tested.
Compute the SHA-1 hash of file X.
Decrease the core voltage by 10 mV.
Did the system crash?
No
Yes
Record the current voltage offset for the current frequency.
Reset the core voltage offset to zero.
Have we recorded 3 voltage offsets for this frequency?
No
Yes
Let V be the maximum (least negative) of the 3 recorded voltage offsets, plus 10 mV.
Set the core voltage offset to V.
Compute the SHA-1 hash of file X.
Did the system crash?
No
Decrease the core voltage by 1 mV.
Yes
Record the current voltage offset as a critical voltage for the current frequency.
Have we recorded 10 critical voltages for this frequency?
Yes
No

Figure 3.1: *Flowchart describing how we collect data to determine the critical points.*

In particular, for each frequency, we first find the critical voltage to the nearest 10 mV, performing three (3) repetitions. We assume that the critical voltage for the current frequency is bounded above by the maximum (least negative) of these three samples plus 10 mV. We then find the critical voltage to as much accuracy as is possible (i.e. to the nearest 1 mV) by testing every possible voltage offset below this upper bound, performing ten (10) repetitions. In Figure 3.2, we plot the mean, minimum, and maximum of these 10 new data points for each possible frequency, which collectively represent what the critical voltage is for that frequency.

Observe in Figure 3.2 the clear trend between frequency and the highest unstable voltage offset for a given frequency. A trend line drawn for this dataset would separate the graph into two regions:

- the upper-left region, which contains all the stable OPPs; and
- the lower-right region, which contains all the unstable OPPs.

3.3 Demonstrating the viability of power management attacks

In order to demonstrate that a power management attack is possible on this platform, we attempt to induce a computational fault whilst the SHA-1 hash

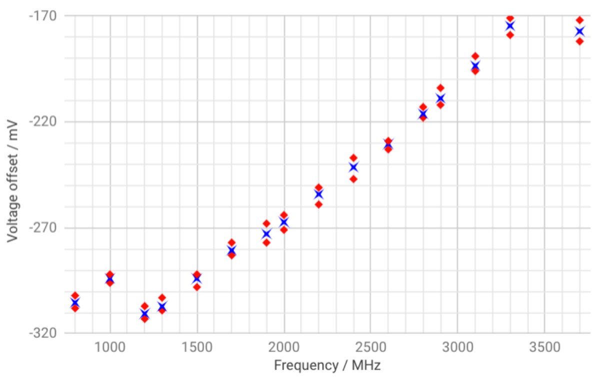


Figure 3.2: *Graph plotting highest unstable voltage offset against frequency, using the data collected in determining the critical points. The blue cross for a given frequency is the mean of the voltage offsets recorded for that frequency. The red diamonds are the bounds for the corresponding error bars (maximum and minimum of the recorded voltage offsets).*

of some known data with a known hash is being computed. The aim is for the fault to result in an incorrect hash being computed. The process is as follows:

(1) Choose a critical point.

(2) Put the system into a stable OPP near the chosen critical point.

(3) Begin computing the SHA-1 hash of the known data.

(4) Whilst the hash is being computed, *briefly* put the system into an unstable OPP near the chosen critical point.

(5) The hash computation ends, and we hope to see an incorrect hash.

We use the data collected in §3.2 to choose a stable OPP and an unstable OPP which both use the same frequency and lie near the stable–unstable boundary line. The larger the error bars are near this dividing line for a given frequency, the less clear the distinction between the stable and unstable regions for that frequency. As such, since the data for 2600 MHz has the smallest error bar, we will choose OPPs with a frequency of 2600 MHz. The data we collected shows that the mean critical voltage for 2600 MHz is −230.4 mV. Thus, we decide that the voltage offset of our stable OPP should be greater than (i.e. more positive than) −230.4 mV, and that of the unstable OPP should be less than (i.e. more negative than) −230.4 mV.

In order to perform steps (2) and (4), we need the ability to alter the voltage scaling parameters of the CPU. Using the `undervolt` utility discussed in §3.1.2 will not afford us the timing precision required to execute step (4) briefly enough to induce a fault without completely crashing the system. As such, we opt to write our own program which will alter the voltage scaling parameters in a suitably brief manner, which requires us to be able to write to the necessary MSR as discussed in §3.1.2. This is ultimately done via execution of the `WRMSR` assembly instruction of the x86 architecture [7, Vol. 2, §4.4]. Since this instruction must be executed at privilege level 0 or in real-address mode, we cannot execute it from userspace; we need to do so in kernel mode. Therefore, we write a custom kernel module to set the voltage offset at the necessary times.

We take the approach of simply demonstrating that successful fault injection is possible, rather than developing a fully-formed attack on a program running in userspace. That is, we could develop a malicious kernel module which lies in wait until a hash computation begins in userspace, e.g. an instance of GnuPG's `sha1sum` program begins running. Upon detecting that such a program is running, the kernel module would have to correctly time the execution of step (4) as discussed in [1, §3.5]. Instead, we take the less complex approach of adapting the source code of `sha1sum` [14] into a kernel module which, when inserted into the kernel, performs steps (2), (3), and (4), all in kernel mode. In this way, we can directly add instructions to the `sha1sum` source code to alter the core voltage at the necessary times. The details of the kernel module we developed for this purpose, dubbed `bad_sha`, are given in §3.4.2. We experiment with the following parameters:

- the voltage offset of the chosen stable OPP;
- the voltage offset of the chosen unstable OPP; and
- the amount of time in which the unstable OPP is used;

and we see that we can successfully induce a fault around 2% of the time when we use −225 mV for the stable OPP, −234 mV for the unstable OPP, and the unstable OPP is in use whilst five (5) 32-bit W-blocks are processed during the SHA-1 hash computation (where a W-block is defined as in [15, §6.1]). When a fault does occur, the specific incorrect result observed is not consistent; there were no two instances where the same incorrect hash was observed.

3.4 Tools

3.4.1 Shell script

The Bash script used in §3.2 to assist us in collecting data to determine the stability of OPPs is given in Listing 3.1.

Listing 3.1: *Shell script to determine critical points — `undervolt-test.sh`*

```bash
#!/bin/bash
cpu_frequency="$1"

all_flag=0
if [[ $# -eq 2 && $2 == "all" ]] ; then
 all_flag=1
fi

voltages_list="undervolt-list-${cpu_frequency}.txt"
if [ $all_flag -eq 1 ] ; then
  voltages_list="undervolt-list.txt"
fi

cpupower frequency-set -f "${cpu_frequency}MHz"
while IFS= read -r voltage; do
  (
    undervolt --core "$voltage" --cache "$voltage"
    shasum -b -a 1 ._shatest_data | (echo -n "$voltage " && cat)
    undervolt --core 0 --cache 0
  )
done < "$voltages_list"
```

The script takes the frequency to test in MHz as its first argument, e.g. `2600` for 2600 MHz, and optionally takes a second argument, `all`. If `all` is specified, we test all voltage offsets listed in the file `undervolt-list.txt`, which simply lists all the multiples of 10 in order from −10 to −400, as can be seen in Listing 3.2 for clarity.

If `all` is not specified, a list of voltage offsets specific to the given frequency is used, as found in the file `undervolt-list-`*f*`.txt`, where *f* is the given frequency. For example, specifying `2600` as the frequency and not specifying `all` will result in the file `undervolt-list-2600.txt` being used as the list of voltage offsets to test. Such a file will list *all* the integers in decreasing order from the upper bound we determined for the critical voltage for that frequency (as discussed in §3.2) expressed in mV, to −400. As an example for clarity, `undervolt-list-2600.txt` is given in Listing 3.3.

The voltage offsets given in these files are expressed in millivolts, as they are passed directly to the `undervolt` utility, which expects voltage offsets in these units, as explained in §3.1.2.

Listing 3.2: *Non-granular list of voltage offsets to test for all frequencies, in millivolts — excerpt of `undervolt-list.txt`*

```
-10
-20
    // ... snipped 36 lines for brevity ... //
-390
-400
```

Listing 3.3: *Granular list of voltage offsets to test for 2600 MHz, in millivolts — excerpt of `undervolt-list-2600.txt`*

```
-220
-221
-222
    // ... snipped 175 lines for brevity ... //
-398
-399
-400
```

3.4.2 Kernel module

The kernel module used in §3.3 to briefly put the system into an unstable state whilst a SHA-1 hash is being computed is adapted directly from the source code of GnuPG's `sha1sum` program [14]. Those functions which we have altered or added are given in Listing 3.4; note that the `main` function has been renamed to `main_routine`. The functions used to write to the MSR in order to perform voltage scaling are in the header file `msr.h`, which is given in Listing 3.5.

We first set the frequency to 2600 MHz via `cpupower`. As can be seen in the `transform` function on lines 23–29 of Listing 3.4, we then alter the voltage offset so that the system is using an unstable OPP, wait until five (5) W-blocks are processed via the `R` function (whose definition can be found in [14]), and then restore the prior voltage offset so that the system is once again using a stable OPP. In order to ensure that the `transform` function is called precisely once so that this brief period of instability only occurs once during the hash computation, the data whose SHA-1 hash we compute must be no more than 512 bits in length. We generate such a bitstring and store it in the file `/tmp/._shatest_data`, which is the file that our kernel module computes the hash of.

Listing 3.4: *Altered code of `sha1sum.c` as used in the `bad_sha` kernel module — excerpt of `bad-sha.c`*

```
1  /* SHA-1 code is taken from GnuPG 1.3.92 */
2  #define KERN_MSG KERN_ALERT KBUILD_MODNAME ": "
3
4  /* Voltage offset of chosen stable OPP, in units of 1/1024 volts. */
5  #define STABLE_VOLTAGE (-230)
6  /* Voltage offset of chosen unstable OPP, in units of 1/1024 volts. */
7  #define UNSTABLE_VOLTAGE (-240)
8
9  #include <linux/kernel.h>
10 #include <linux/module.h>
11 #include <linux/printk.h>
12 #include <linux/fs.h>
13
14 #include "msr.h"
15
16 /* Transform the message X which consists of 16 32-bit-words */
17 static void transform(SHA1_CONTEXT *hd, unsigned char *data)
18 {
19         // ... snipped 49 lines for brevity ... //
20     R(a, b, c, d, e, F1, K1, x[0]);
21         // ... snipped 31 lines for brevity ... //
22     R(d, e, a, b, c, F2, K2, M(32));
23         write_vcore_dec(UNSTABLE_VOLTAGE);
24     R(c, d, e, a, b, F2, K2, M(33));
25     R(b, c, d, e, a, F2, K2, M(34));
26     R(a, b, c, d, e, F2, K2, M(35));
27     R(e, a, b, c, d, F2, K2, M(36));
28     R(d, e, a, b, c, F2, K2, M(37));
29         write_vcore_dec(STABLE_VOLTAGE);
30     R(c, d, e, a, b, F2, K2, M(38));
31         // ... snipped 40 lines for brevity ... //
32     R(b, c, d, e, a, F4, K4, M(79));
33
34     /* Update chaining vars */
35     hd->h0 += a;
36     hd->h1 += b;
37     hd->h2 += c;
38     hd->h3 += d;
39     hd->h4 += e;
40 }
41
42 /* Get the lowercase hexadecimal digit correponding to a given number that is between 0 and 15 (inclusive). */
43 char digit_to_hex(int n)
44 {
45     return n < 10 ? '0' + n : 'a' + n - 10;
46 }
47
48 /* Put the hash described by the given SHA-1 context `ctx` into the string `hash`. `hash` must point to a buffer with sufficient memory allocated (at least 41 bytes). */
49 void hash_from_context(char *hash, SHA1_CONTEXT ctx)
50 {
51     int i;
52     unsigned char byte;
53
54     for (i = 0; i < 20; i++) {
55         byte = ctx.buf[i];
56         hash[2*i    ] = digit_to_hex(byte / 16);
57         hash[2*i + 1] = digit_to_hex(byte % 16);
58     }
59     hash[40] = '\0';
60 }
61
```

```c
/* Compute the SHA-1 hash of the file at path `filename` and print it via `printk()`. */
int main_routine(char* filename)
{
    struct file *fp;
    unsigned char buffer[4096];
    ssize_t n;
    SHA1_CONTEXT ctx;
    loff_t file_offset = 0;
    char hash[41];

    fp = filp_open(filename, O_RDONLY, 0);
    if (IS_ERR(fp)) {
        printk(KERN_MSG "can't open '%s'\n", filename);
        return -1;
    }
    printk(KERN_MSG "Opened file successfully\n");

    sha1_init(&ctx);

    while ((n = kernel_read(fp, buffer, sizeof buffer, &file_offset)) > 0) {
        sha1_write(&ctx, buffer, n);
    }
    if (n < 0) {
        printk(KERN_MSG "error reading '%s'\n", filename);
        return -1;
    }
    // assert n == 0, so we have reached end of file
    sha1_final(&ctx);
    filp_close(fp, NULL);

    hash_from_context(hash, ctx);
    printk(KERN_MSG "%s  %s\n", hash, filename);

    return 0;
}

static int __init mod_init(void)
{
    write_vcore_dec(STABLE_VOLTAGE);
    printk(KERN_MSG "Inserting module\n");
    return main_routine("/tmp/._shatest_data");
}
module_init(mod_init);

static void __exit mod_exit(void)
{
    write_vcore_dec(STABLE_VOLTAGE);
    printk(KERN_MSG "Removed module\n");
}
module_exit(mod_exit);
```

Listing 3.5: *Header file containing MSR-related functions* — `msr.h`

```c
#include <linux/kernel.h>

/* Write the value `data` to the MSR `msr` via the assembly instruction WRMSR. */
static void write_msr(u32 msr, u64 data)
{
    u32 low = (u32)data;
    u32 high = (u32)(data >> 32);

    asm volatile("wrmsr" : : "c"(msr), "a"(low), "d"(high));
}

/* Set the core (and cache) voltage offset as described by `hex_offset`, which is a 16-bit value whose lowest 11 bits encode an 11-bit signed integer whose value is the desired core voltage offset in units of 1/1024 volts. */
static void write_vcore_hex(u16 hex_offset)
{
    write_msr(0x150, 0x8000001100000000 | ((u32)hex_offset << 21));
        // plane index = 0, write flag = 1
    write_msr(0x150, 0x8000021100000000 | ((u32)hex_offset << 21));
        // plane index = 2, write flag = 1
}

/* Set the core (and cache) voltage offset to `signed_offset`, which is in units of 1/1024 volts. The offset must not exceed +/-1 volt. */
static void write_vcore_dec(s16 signed_offset)
{
    u16 hex_offset = signed_offset < 0 ? signed_offset + 2048 : signed_offset;

    if (hex_offset & ~(~0 << 5) << 11) {
        printk(KERN_MSG "The value %d is not an 11-bit signed integer!\n", signed_offset);
        return;
    }

    write_vcore_hex(hex_offset);
}
```

4 Conclusion

4.1 Summary and discussion

In summary, we have been able to meet our goal of successfully demonstrating that hardware-based power management mechanisms (specifically DVFS) present in an Intel Core desktop processor (namely a 4th generation Haswell unit) that are exposed via software can be exploited in order to affect the result of a computation. Though our observations are infrequent, they are undeniably present, and the ubiquity of software-exposed DVFS features in these processors ultimately means that the proof of concept detailed in §3.3 should be easily portable to any Intel Core processor manufactured to date; if a proper attack can be developed that makes use of this attack vector, then all Intel Core processors should be vulnerable to that attack. Moreover, the specific processor used in our experimentation is incapable of per-core DVFS as discussed in §2.1, demonstrating for the first time that a power management attack is viable on such hardware.

With regard to SGX, Intel's secure execution environment, this remains to be explored proper, but it is reasonable to assume that programs exclusively using memory in an SGX enclave are just as vulnerable as those that do not take advantage of SGX at all. This is because the faults we have induced occur within the CPU itself, and not in memory, resulting in logic being carried out within the CPU being affected; the content of CPU registers themselves are likely to be made erroneous, resulting in the observed computational errors.

We remark that the data collection carried out in §3.2 and the experimentation conducted in §3.3 could be improved upon in hindsight. The decision to determine critical voltages only to the nearest 1 mV rather than the highest possible degree of accuracy, namely to the nearest $\frac{1}{1024}$ volts, was made due to the then-assumed necessity of using the `undervolt` utility discussed in §3.1.2 to perform voltage scaling. Since developing the kernel module discussed in §3.4.2, it is now understood that this data collection could be done with greater precision. With regards to determining the optimal values to produce an observable fault, this ought to have been done in a more systematic manner, which would have been possible given the time to develop the necessary tools.

4.2 Future work

As discussed in §2.1, prior research in this area has demonstrated complete attacks using power management as an attack vector. We would like to have explored this ourselves and develop the necessary kernel module as discussed in §3.3 which would prey on programs running in userspace to achieve the same effect as we have already observed.

Once such a kernel module is developed, we would also like to explore SGX specifically, as discussed in §4.1, attempting to observe the same results once again, but with a program that makes exclusive use of memory in an SGX enclave as the victim. This would require access to a 6th generation Skylake or later Intel Core processor, as SGX is not present on earlier units. If such an observation is made, the natural next step is to replicate the AES key extraction attack in [1] on this platform, with the AES decryption program running exclusively in an SGX enclave.

More recent processors using the x86 instruct set implement specific instructions for efficiently computing the results of AES rounds. These additional instructions are known as the AES New Instructions (AES-NI) [16]. If the AES key extraction attack is possible on devices using Intel Core processors that support SGX, it is in our interest to explore whether the AES key extraction attack in [1] is still possible if AES-NI instructions are being employed.

Bibliography

[1] Adrian Tang et al. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *Proceedings of the 26th USENIX Security Symposium*. 2017. ISBN: 978-1-931971-40-9. URL: `https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-tang.pdf` (visited on 04/08/2019).

[2] Yoongu Kim et al. *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors*. 2014. URL: `http://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf` (visited on 04/08/2019).

[3] Sheng Zhang, Adrian Tang et al. *Blacklist Core: Machine-Learning Based Dynamic Operating-Performance-Point Blacklisting for Mitigating Power-Management Security Attacks*. 2018. URL: `http://0x0atang.github.io/files/islped18_blcore.pdf` (visited on 04/08/2019).

[4] Yehuda Lindell. *The Security of Intel SGX for Key Protection and Data Privacy Applications*. 2018. URL: `https://cdn2.hubspot.net/hubfs/1761386/security-of-intelsgx-key-protection-data-privacy-apps.pdf` (visited on 04/08/2019).

[5] Michael Tunstall et al. *Differential Fault Analysis of the Advanced Encryption Standard using a Single Fault*. 2009. URL: `https://eprint.iacr.org/2009/575.pdf` (visited on 04/08/2019).

[6] Rob Landley and the Linux Kernel Organisation, Inc. *CPU Performance Scaling — The Linux Kernel Documentation*. 2019. URL: `https://www.kernel.org/doc/html/v5.0/admin-guide/pm/cpufreq.html` (visited on 04/08/2019).

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2019. URL: `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf` (visited on 04/08/2019).

[8] Rob Landley and the Linux Kernel Organisation, Inc. *intel_pstate CPU Performance Scaling Driver — The Linux Kernel Documentation*. 2019. URL: `https://www.kernel.org/doc/html/v5.0/admin-guide/pm/intel_pstate.html` (visited on 04/08/2019).

[9] ArchWiki. *CPU frequency scaling*. URL: `https://wiki.archlinux.org/index.php/CPU_frequency_scaling` (visited on 04/08/2019).

[10] Intel Corporation. *Intel® Core^{TM} i5-4590 Processor Product Specifications*. URL: `https://ark.intel.com/content/www/us/en/ark/products/80815/intel-core-i5-4590-processor-6m-cache-up-to-3-70-ghz.html` (visited on 04/08/2019).

[11] Username `unclewebb`. *The ThrottleStop Guide — Notebook Review*. 2010. URL: `http://forum.notebookreview.com/threads/the-throttlestop-guide.531329/` (visited on 04/08/2019).

[12] Miha Eleršič. *mihic/linux-intel-undervolt: Guide to linux undervolting for Haswell and newer Intel CPUs*. 2017. URL: `https://github.com/mihic/linux-intel-undervolt` (visited on 04/08/2019).

[13] George Whewell. *georgewhewell/undervolt: Undervolt Intel CPUs under Linux*. 2017. URL: `https://github.com/georgewhewell/undervolt` (visited on 04/08/2019).

[14] Free Software Foundation, Inc. and g10$^{\text{CODE}}$ GmbH. *sha1sum.c - GnuPG 1.3.92*. 2001. URL: `ftp://ftp.gnupg.org/gcrypt/binary/sha1sum.c` (visited on 04/08/2019).

[15] IETF. *RFC 3174 - US Secure Hash Algorithm 1 (SHA1)*. 2001. URL: `https://tools.ietf.org/html/rfc3174` (visited on 04/08/2019).

[16] Jeffrey Rott, Intel Corporation. *Intel® Advanced Encryption Standard Instructions (AES-NI)*. 2012. URL: `https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni` (visited on 04/08/2019).