Plundervolt Library: An Environment for Hardware and Software Undervolting

Cyril Saroch
School of Computer Science, University of Birmingham, UK

1. Abstract

Recently, a number of new attacks exploiting the Intel SGX system have emerged. Among them most crucially for this paper Plundervolt, and VoltPillager. They outline two forms of intrusion into a system protected from Rowhammer-style attacks, and show the way Intel has chosen to remedy its vulnerabilities to such attacks is insufficient.

Both papers explain how voltage control during code execution can render these fixes useless. In this report, we aim to provide a C library for working with such attacks, with the goal to ease the exploration of Intel's solutions to better understand and fix the issue.

Rather than experiment with the attacks ourselves, we have chosen to implement a comprehensive, simple-to-use, and well-documented system, and we hope to thus improve the way others will work with Plundervolt and VoltPillager, or any other voltage-based exploits.

We will discuss our initial decisions when creating the library, our design choices and their outcomes, and explain how the library achieves our aims, and how it helps with writing new code based on voltage control. We provide comprehensive lists of available functionalities and example codes, as well as points to be wary of while using the library.

Additionally, we talk about cases where our predictions failed, and where the library lacks as a result.

We have omitted any discussion of the potential of voltage scaling attacks, as we feel it best left for specifically aimed studies. Instead, we only explain and offer a way of implementing them.

Contents of the Paper

2. Introduction and Background

Several papers (1–5) have described attacks on virtually any modern Intel CPU model via controlling the voltage accessible to the processor. In this paper we describe a library, which we have written to ease the process of coding these attacks.

The attacks' goal is to introduce a fault to a piece of code running in the CPU, and while there are other uses for this attack (6,7), the original objective was to fault the Intel Secure Enclaves.

We will firstly describe these enclaves, and why it is important to search for vulnerabilities in them, and then outline the basic ideas of the attacks.

2.1. Intel SGX

After papers such as Rowhammer (8), which showed that purely software-based attacks can exploit an otherwise secure system by repeatedly accessing neighbouring memory cells, Intel resolved this problem by introducing Intel SGX.

Rowhammer targets physical sections of memory to access or change data in DRAM. Intel deals with this issue in several ways (9,10), most vitally for us by providing Software Guard Extensions (11) (SGX) - software (usable by the operating system and users alike), which allocates non-addressable memory regions in RAM called (secure) *enclaves*.

Functions can be called inside of the enclave, and data is stored there, but only the results of those functions can be read from the outside. No process other than a function running inside of the enclave has access to data also stored in it.

This should in theory prevent attacks like Rowhammer to repeatedly access memory cells around the attacked region, and thus combat any such vulnerabilities. To the best of our knowledge, this has been successful, even though the mitigation is not perfect (12).

An important part of this scheme is that nobody has control over the data in an enclave. Not even the root user can reach it. Intel SGX should therefore be secure against a user with sudo privileges.

2.2. Attack Model

Since Intel SGX ought to be secure against attackers with physical access to the computer and sudo privileges, the attack model can assume a root user as the adversary, and still produce a significant, dangerous attack. Both versions of the voltage-based attack (software and hardware undervolting) require sudo access.

The adversary needs a way to fix the CPU frequency to a constant value. This can be done with a simple piece code, such as is shown in Figure 1 |1|.

```
#!/bin/bash
sudo cpupower -c all frequency-set -u $1
sudo cpupower -c all frequency-set -d $2
sleep 1
```

Figure 1: Bash script for setting a constant frequency.

The adversary must have a way to manipulate the voltage. This can be done through a model-specific register (MSR) – a device, as explored by the Plundervolt paper (software undervolting), or via direct, physical access to the voltage control system, described by VoltPillager (hardware undervolting). Both strategies will be explained in their respective sections below |Software Undervolting, Hardware Undervolting|.

If the adversary uses hardware undervolting, they need physical access to the computer to install a micro-controller, which will send signals to the voltage control system. However, once installed, the attack may be performed remotely.

Finally, we assume a Linux operating system.

2.3. Software Undervolting

First, we will look at software undervolting as proposed by the Plundervolt paper (1). The idea of the attack is simple: While a computation is being performed, we lower the voltage to critical levels. This, when done with the right parameters, causes an error in the computation.

A function inside a secure enclave will use securely stored data. Neither of these is susceptible to a direct attack, but the function's computation occurs inside the CPU, not the enclave. The wrong result, caused by undervolting, will be written to secure memory. Intel SGX has no way of stopping this.

The voltage scaling system is built in to Intel power management (13). Voltage depends on frequency, and is computed as a pair of the two. When the CPU requires higher frequency, it asks for it, and with higher frequency the voltage rises, too. Conversely, lower frequencies need lower voltages.

Before performing the attack, the adversary sets the voltage as shown in Figure 1 |1|. The CPU will not change the voltage on its own, but through access to MSR, which houses the interface for recomputing it. The adversary has access to MSR, too.

By sending correctly formatted byte streams (5,14,15) to the right offset of MSR, the voltage will be changed automatically. As a computation takes place, the voltage is lowered in this way to the desired value. It is tricky to find the right voltage value, since too high will not cause a fault, and too low will freeze the computer altogether. More on this later |Limitations, Reflection|.

Some, simple operations, such as addition or bitwise XOR, cannot be undervolted. However, multiplication can be successfully undervolted with reliable results, i.e., our experiments |Practical Results| show the result of a faulty multiplication will be the same regardless on the CPU model, as long as the multiplicands are the same. It stands to reason more complicated operations are also vulnerable to undervolting.

Finally, since every Intel Core x84 processor has this MSR interface, the adversary only needs to set the frequency, initiate the voltage change at the right time, and the computer will take care of the rest. After the attack, the voltage may be just as easily returned to its safe levels |Resetting the Attack|.

2.4. Hardware Undervolting

As with software undervolting, the aim of this attack is to cause an error in computation due to lower voltages than are necessary for a static frequency. However, unlike software undervolting, this attack interferes with the low-level realities of the computer to do so.

When a new voltage is requested, a bus called Serial Voltage Identifier (SVID) carries the signal to the Voltage Regulator. This is a separate chip on the motherboard, which changes the voltage as requested. These signals are not encrypted, and can be reproduced.

We take advantage of the reverse engineering done in the VoltPillager (2) paper, and work with their suggested design based on the micro-controller Teensy 4.0.

Teensy is connected to the Voltage Regulator. When the adversary wants to run the attack, they send the configuration to the micro-controller via a USB connection. Teensy sends instructions to the Regulator, pretending to be the CPU asking for lower voltage (or higher, if the adversary is *overvolting*). Since the signal is not encrypted, the Voltage Regulator does not differentiate between a real and simulated request, and so performs the change.

This paper will illustrate the differences in usage between software and hardware undervolting. There are many such differences, but the results are comparable. In other words, the same attack will produce, possibly at different voltages, the same faulty result.

3. Design and Implementation

As stated above, the goal of this project was to create a library facilitating a simplified implementation of the two voltage-based attacks.

In this part, we will look at how this was achieved, what challenges we faced, and what the final result looks like.

The library consists of two files - `plundervolt.h` and `plundervot.c`. We chose the name Plundervolt (1), but it implements code from VoltPillager (2) as well. In addition to these two, there is a sub-directory `arduino`, which was lifted verbatim from VoltPillager, as it is necessary to communicate with Teensy. We will mention it later |Arduino Library|.

3.1. Library Structure

When designing the library, we separated functions into four broad groups:

1. Private functions
   These functions are usually not necessary. They exist only for readability and convenience while coding. They implement small tasks called by other, public functions.
2. Software focused functions
   These functions take care of software undervolting. They write to the MSR module, or return the current undervoltage (i.e., how many *mV* the system is below the base voltage for the set frequency).
   As these functions are meant to be accessed by the user, they are all public.
3. Hardware focused functions
   Similar as above, but with hardware undervolting. These functions communicate with Teensy (via the Arduino library) through a USB connection, as well as with an onboard trigger, when told to do so. This trigger is another way of talking to Teensy, and is time-wise more accurate.
   These functions are also public.
4. Maintaining and convenience functions
   These functions are not necessary to run the library, but may help with it, or take care of a general task required by both versions of undervolting.
   These for instance open files (MSR,…), run user-provided functions in or outside of loops and in their separate threads, help with configuring Teensy in the right order, or help with the entire attack process in a likely sequence of function calls.
   Some of these functions, such as `plundervolt_apply_undervolting()`, do not need to be called at all. The user can do everything in them on their own by calling the public functions they need.
   Some of these are public, others private.

Beyond these functions, there is a list of error codes, described below, and a single structure, which holds all parameters of the attack (voltage, duration of the attack,…) |Setup, Appendix B - List of Undervolting Parameters|. This is discussed in the following section.

3.2. Setup

The library is set up with attack parameters through a structure `plundervolt_specification_t`. It holds variables for both software and hardware undervolting, as well as an *enum* type `undervolting_type`, which tells the library what version of undervolting to perform.

The function `plundervolt_init()` creates this structure with default values, and returns it. Before doing any undervolting, the user should change what variables they deem appropriate, and confirm their choice with `plundervolt_set_specification()`. These are necessary steps, illustrated in Figure 2 |2|.

```
// Create the structure.
plundervolt_specification_t specification;
// Fill the structure in.
specification = plundervolt_init();
// Set some parameters.
specification.u_type = hardware;
specification.wait_time = 300;
// Confirm your parameters.
plundervolt_set_specification(specification);
```

Figure 2: Creating and changing the library specification structure.

By doing this, a measure of security is enforced onto the library. A series of checks is performed by `plundervolt_set_specification()` to ensure the parameters make sense. These checks can be done by the user as well, by calling `plundervolt_faulty_undervolting_specification()`, but only *after* confirming the specification, otherwise the user may believe the confirmation took place, when in fact they only checked it was sensible.

Some variables are universal for both types of undervolting, e.g. `u_type` (the version of undervolting), and `wait_time` (number of *ms* to sleep for in strategically chosen stages of the attack). Beyond that, a function pointer for the target code to introduce fault into, as well as its arguments, is necessary in some cases (such as using `plundervolt_run()` |3.5.4|). Conversely, a function pointer (and its arguments) for the function which stops the undervolting, is optional in all cases. Other parameters are focused on software and hardware only.

A full list of parameters (and their brief explanations) can be found in Appendix A.

3.3. Software Undervolting

First, we will focus on software undervolting functions.

Before starting the attack, the library opens the MSR module. It acts as a standard file, though it needs root privileges to be accessed.

Two instructions must be sent to MSR. Both contain the new voltage, but differ in some constants understood by the module. The public function `plundervolt_software_undervolt()` performs both writes, but `plundervolt_set_undervolting()` may be used twice in succession instead. To compute the right format, it uses `plundervolt_compute_msr_value()`. The latter two functions are lifted from the Plundervolt (1) paper, as one is a mathematical operation calibrated to speed, which is essential, and the other is a simple `pwrite()` function call. There is little to change.

As the voltage changes, a function, which we will call "undervolted function", executes, and may experience faulty computations. While this can be done consecutively, we would recommend having the undervolted function running in its own thread "function thread", and create another thread "undervolting thread", which writes to the MSR. It is possible to have multiple function threads. All undervolted functions will run in parallel, and any one of them can experience the fault.

We have written the library in such a way that both approaches are possible. However, we prefer the thread strategy to the consecutive one, as that increases the chances of a fault, so most convenience functions work best with that.

The user may wish to have absolute control over the process. In this case, they would call each function themselves in the right order, without using convenience functions as `plundervolt_apply_undervolting()` or `plundervolt_run()`. In this case, the user must take care of the threads, which includes creating and joining them.

When using `plundervolt_apply_undervolting()`, which performs the voltage change, it must be running in its own thread already. The function `plundervolt_run()` creates those threads, among many other things, which will be discussed later |Overarching Convenience Function|.

When using multiple threads, we had to make sure the user had the option of stopping the whole process from any one of them. The solution was a global variable accessible by `plundervolt_set_loop_finished()`. This variable is used in two types of checks throughout the library. Firstly, if undervolting happens with incrementally changed values, that is in a loop where each iteration changes the voltage, this loop is stopped.

Secondly, the user could tell the library to run the undervolted function (in each of the function threads) multiple times. The global variable stops these loops as well, in every function thread.

Effectively, `plundervolt_set_loop_finished()` acts as a signal, which is sent to all threads controlled by the library, and which stops the operations in them. This is a powerful, yet easy method, which can be coupled with another useful functionality of the library. The user provides the undervolted function as a pointer, but there is another function pointer to the "checking function".

Imagine the undervolted function is called in a loop. If everything runs smoothly, the function finishes, and is called again. When a fault occurs, the user wishes to stop this loop, and the undervolting altogether. How is this done?

Either there is a check inside of the undervolted function, which calls `plundervolt_set_loop_finished()`, or the library calls the checking function on every iteration. This function then checks whatever condition the user wants (not necessarily a faulty computation), and causes the process to stop. The default pointer is NULL, which means there is no checking function provided.

But be aware, unless the library specification explicitly says "There is no checking function" (see Appendix B for the parameter `integrated_loop_check`), even a NULL pointer function will be executed (with a runtime error). The reason for this is described in detail in the section Limitations.

In addition to these functionalities, the library can also return the value of current undervolting by calling `plundervolt_read_voltage()`, which is likewise lifted from the Plundervolt (1) paper, since it is fairly simple, and requires numerical parameters discussed in said paper.

3.4. Hardware Undervolting

As with software undervolting, before running any attacks, files must be open. This time, however, they are not the MSR module, but devices for communicating with the micro-controller (Teensy in our case), and optionally the onboard trigger.

Since these are not built in modules, they must be specified with a string variable (stored in the library specification). Through these, the library sends instructions to Teensy, and receives responses from it, which are printed on screen whenever possible.

In addition to opening the files, the initialisation of hardware undervolting requires some other settings. These are copied from the VoltPillager (2) paper, as they are very specific.

Before the attack is commenced, Teensy must receive specification for it. All the variables (staring voltage, attack voltage, end voltage, duration of each of those steps, and so on) are provided through the library specification |Setup, Appendix B|.

The library uses a public function `plundervolt_configure_glitch()`, which sends all the specification to Teensy. The user may use this function on their own, or let one of the convenience functions do it (`plundervolt_apply_undervolting()`, which in this case only prepares is, or `plundervolt_run()` |Overarching Convenience Function|).

After configuration, the attack must be "armed". This tells Teensy to prepare to undervolt, but wait for a signal. The public function `plundervolt_arm_glitch()` is used here.

Unlike software undervolting, where the attacker sets the voltage, and it stays that way, hardware undervolting must be started, then stops on its own after time specified by the attacker in the configuration (in *mV*).

When `plundervolt_fire_glitch()` is called, the attack starts. Either a newline is sent to Teensy, signalling the end of input, or the onboard trigger, in contact with Teensy, is activated, which has the same effect. There is a delay (naturally chosen by the user), after which the starting voltage is set. Then, after some time (also user-chosen), we fix the attacking voltage.

Due to this difference between the attacks, hardware undervolting doesn't simply write to Teensy and let it work, but prepares it (configures and arms it), then waits for the user to start the attack *inside of the undervolted function*. This is also done for greater time control.

Because the undervolted function also starts the attack, having two functions do so in parallel and end the cycle at different times made less sense. For that reason, we judged threads inefficient for hardware undervolting, although they can be implemented easily by the user, of course.

The attack stops after the given time, and the trigger should be reset, if used. This must be done by the user, as the library doesn't know when the attack ends. Use `plundervolt_reset_voltage()` to do this - more on this function in Section Resetting the Attack.

A new attack must start with the configuration step.

Arduino Library

The VoltPillager (2) paper used a modified version of the Arduino library (16) to communicate with Teensy. Since we do not mean the user to need it (see the previous comments about abstraction), its functions remain private.

It was included for our convenience alone, as all its functions would have to be written again, in much the same form.

3.5. Other

The following are concepts and features too short to deserve their own sections.

Errors

The library has a number of defined error codes. Each function, which has the potential to fail in some way, returns a `plundervolt_error_t` code.

Since `PLUNDERVOLT_NO_ERROR` is equal to 0, the return value of the functions may be used as a boolean value.

To print the generic error details, use `plundervolt_print_error()` and pass the error to it. It uses `plundervolt_error2str()` to get the string.

Cleanup

After the library is finished, all files must be closed. The user should call `plundervolt_cleanup()` (in both versions of undervolting). No convenience function does this, as the library cannot know if the user will continue or not. This is the last public function that should be called.

Safe Specification

As mentioned above, the library provides a way to check the specification with `plundervolt_faulty_undervolting_specification()`. In cases such as giving no undervolted function, or not pointing to the checking function, an appropriate error is returned.

Overarching Convenience Function

There is a previously discussed function `plundervolt_run()`. We have anticipated the most likely usage of the library, and written a function which does as much as possible for the user. This library opens the files, creates threads if appropriate (based on the type of undervolting, and the number of threads asked for), calls the undervolted function in each of the function threads, and starts undervolting in the undervolting thread.

It runs the user-defined functions, in loops or on their own, undervolts while doing so, and lowers the voltage (if software-undervolting) from a starting value to an end value. It also calls the checking function, to stop the loops and undervolting,.

Alternatively, it configures Teensy, arms it, and calls the undervolted function (possibly in a loop), which then fires the glitch (starts the attack). All this, starting with configuration, is also a big loop, repeated as many times as the user wishes (see Appendix B for `tries`).

After the attack, it joins the threads (if there are any).

The only thing it does not do is call `plundervolt_cleanup()`, since the user may want to continue using the library afterwards.

Passing Arguments

There is a simple way to pass arguments to the `function` (undervolted function) and `stop_loop` (checking function) function pointers. As these two functions are implemented by the user, along with their argument lists, the library must not assume any argument types.

To deal with passing arguments to these functions when they are called by the library (such as `plundervolt_run()` would do), we provide `arguments` and `loop_check_arguments` respectively. These are void pointers. Each is meant to point to a user-defined structure (or a single argument of any king), cast to a void pointer.

It is the responsibility of the user to write the functions in such a way that a void pointer is passed to them, and interpreted. An example is provided in Figure 3 |3|.

```
typedef struct argument_structure {
  int a;
  int b;
} argument_structure;

int add (void* arguments) {
  // Cast to arguments_structure pointer
  argument_structure* arguments_pointer =
(argument_structure *) arguments;

  // Access structure
  int a = arguments_pointer->a;
  int b = arguments_pointer->b;

  return a + b;
}

void main() {
  // Create the structure
  argument_structure arguments;
  arguments.a = 1;
  arguments.b = 2;
  // Cast to void pointer
  void* arguments_pointer = (void *) &arguments;

  add(arguments_pointer); // Expecting 3
}
```

Figure 3: Passing arguments to function pointers.

It is also possible to not have any arguments at all, as is the case in the examples we have given |Example Usage|. In this case, the library uses the default NULL value for both argument pointers. This will not cause an error.

Resetting the Attack

After software undervolting ends, the voltage does not change. To return to normal values, use `plundervolt_reset_voltage()`. Note that `plundervolt_run()` does this step, too.

In hardware undervolting, the voltage is reset automatically. However, if the onboard trigger was used, we must reset it. Use the same function to do so. If the trigger was not used, and this function is called, nothing happens.

4. Evaluation and Examples

4.1. Example Usage

To illustrate the library's functionality in action, we provide two example use cases, one for each undervolting type. They both perform the same operation - a faulty multiplication attack, but with key differences to stress how the usages differ between software and hardware undervolting.

In both cases, we set a high voltage, one which we judged to be safe on our machines. Note that these attacks were performed on different computers with the same results.

We set the voltage, and run a loop of two multiplications per iteration. We compare the results, which ought to be the same. As long as they are, we lower the voltage, i.e., increase the undervolting, and try again. We repeat this process until the point of fault.

Both files employ the same strategy for dealing with errors. However, we do not check in the same places to illustrate this is not necessary. An example is shown in Figure 4 |4|.

```
plundervolt_error_t error_maybe =
plundervolt_set_specification(spec);
if (error_maybe) {
  plundervolt_print_error(error_maybe);
  return -1;
}
```

Figure 4: Receiving and printing errors from the library.

Directly after setting the specification, in the way we have shown in Figure 1 |1|, we call `plundervolt_run()`. Here the versions start to diverge.

Software undervolting creates a given number of function threads, and runs the provided function `multiply()`. Since this function calls `multiplication_check()`, we must set `integrated_loop_check` to 1 (meaning the function calls `plundervolt_set_loop_finished()` itself, and doesn't need the library to do it by calling the checking function. The `multiply()` function is called repeatedly (hence `spec.loop = 1`), until the loops are stopped.

We set the starting undervoltage (`start_undervoltage`) to -130 mV, but this should be changed for each computer to an experimentally found "safe" value. The value is negative, since we want to go "130 mV under the base voltage for the given frequency".

After setting all parameters, `plundervolt_run()` starts undervolting with -130 mV undervoltage. It lowers this voltage until it either receives the signal to stop (`plundervolt_set_loop_finished()`), or reaches the end undervoltage (-230 mV). This is the minimal value we felt comfortable setting, as lower voltages would cause a computer crash. This variable is likewise to be set according to each computer.

After the undervolting stops, and all loops are ceased, `plundervolt_run()` joins all threads. After this, `plundervolt_cleanup()` is called, and we are done.

Note that this gradual lowering of voltages is built into the library, as it is a very likely usage. By setting a large `wait_time` value, and setting `end_undervoltage` to `start_undervoltage`, we are effectively saying "only set voltage to this one value, and hold it for `wait_time` number of ms". This is to be done if the user doesn't want to lower the voltage at all, but hold in one place. The voltage is lowered by the parameter `step`.

On the other hand, we have hardware undervolting. The hardware variant of the attack is much more precise, since the adversary communicates with the Voltage Regulator directly. As will be discussed in sections Practical Results and Reflection, it is very hard to find the right parameters, but once it is done, hardware undervolting will work with the same faulting voltage for a while. Hence, coding a system that enforces this lowering of voltages seemed counterproductive.

Instead, what the library does, is prepare the attack with a single voltage, and call the undervolted function. In this case, that is also called `multiply()`. Once again, `multiply()` checks for the fault (and stopping of the whole process) itself, so `integrated_loop_check` is also set to 1, and the checking function can be left out.

We set the two device strings (`teensy_serial` for communication with the micro-controller, and `trigger_serial` for the on-board pin), and define all parameters of a single attack. Note that these parameters will be used over and over, as long as we keep calling `plundervolt_apply_undervolting()`, which in this example is done through `plundervolt_run()`.

When the undervolted function is called, it fires the glitch, and immediately proceeds to perform a loop of multiplications. If a fault is found, all operations are stopped, and `plundervolt_cleanup()` is called. Otherwise, we manually change the `undervolting_voltage` parameter, and try again.

The effect of both programs is the same, but we hope the examples have shown how the library can be used differently. We have not provided more complex examples due to time constraints, and the fact they would be more of a show than a helpful reference, since - by definition - they would be more complicated to understand.

4.2. Practical Results

In this section, we showcase a few examples of the library in action. We have run our example codes discussed in the section Example Usage with different parameters, to see what the results could be.

We have decided to only include a small sample size, as the results are meant to be illustrative of both the library, and the difficulty of setting the correct parameters, and five such examples seemed enough for each undervolting type.

We state the parameters of the attack - the multiplicands, the constant frequency, the number of iterations the multiplication went through on each voltage, the voltage (or undervoltage) at which the fault occurred, and the result of the multiplication.

Software

To begin with, we show |Table 1| the software undervolting attack. This was run on an Intel Core i7-8550U processor (quad-core). Note that as long as the operands are the same, regardless of the fault undervoltage or the frequency, the incorrect result is the same, i.e., the fault behaves predictably.

Our last example did not cause a fault. This was due to the lower number of iterations, which didn't give the library enough time to cause a fault. The fault undervoltage of that entry denotes the undervoltage at which the system *crashed*.

The Fault Undervoltage field shows the difference between the base voltage, and the actual one, not the absolute voltage of the CPU.

Operand 1	Operand 2	Frequency	Iterations	Fault Undervoltage	Correct Results	Incorrect Result
0xAE0000	0x18	2 GHz	1 Billion	-132 mV	c500000	10500000
0xAE0000	0x18	2 GHz	1 Billion	-140 mV	c500000	10500000
0xAE0000	0xF	2 GHz	1 Billion	-141 mV	a320000	ffffffffca320000
0xAE0000	0xF	1.8 GHz	1 Billion	-136 mV	a320000	ffffffffca320000
0xAE0000	0xF	2 GHz	500,000	-209 mV	a320000	-----------

Table 1: Software undervolting results on an i7-8550-U chip.

Hardware

Our hardware undervolting samples |Table 2| are similar to software undervolting. We performed the attack on a different CPU (Intel Core i3-9100) so that we could show the attack works the same, and has the same results.

Unfortunately, we found it much harder to undervolt successfully on this machine. The various changes to the parameters cause not always obvious shifts in execution time, which in turn make it harder for us to set the right parameters. As this is a very common occurrence with voltage scaling attacks, we opted to include these examples.

In the third and fourth entries, as above, the fault voltage is actually the voltage at which the system crashed.

All fault voltages show the absolute voltage of the CPU during the attack.

Operand 1	Operand 2	Frequency	Iterations	Fault Voltage	Correct Results	Incorrect Result
0xAE0000	0x18	3.6 GHz	100,000	0.799 mV	c500000	10500000
0xAE0000	0x18	3.6 GHz	100,000	0.803 mV	c500000	10500000
0xAE0000	0xF	3.6 GHz	100,000	0.783 mV	a320000	--------
0xAE0000	1x18	3.6 Ghz	50,000	0.773 mV	c500000	--------
0xAE0000	0x18	3.8 GHz	100,000	0.789 mV	c500000	10500000

Table 2: Hardware undervolting results on an i3-9100 chip.

4.3. Limitations

We do not make any claim to have written a perfectly functional piece of code. There are a few key points which we believe are noteworthy, and which may negatively impact the usage of our library.

First of all, there is an issue we do not believe could be solved. The library will not, and cannot, work with the same parameters every time, nor can we provide a simple guide to find them for any particular computer. The reason for this is simple - we deal with low-level hardware processes, which must be brought to a very finely tuned point of failure without compromising the whole system. As a result, we are dependant on many outside circumstances. The failure voltage depends on the temperature of the CPU, the number of threads running on it, the load each thread must work with, the CPU and computer models, the imperfections in manufacturing, even the heat of the room.

As a result, the failure voltage changes not only day to day, but hour to hour. We must therefore leave it up to the user to find the right parameters. We would recommend saving every open file before attempting to do so, as a great deal of hard restarts will be needed.

There was the option of writing a simple script for finding the vicinity of the faulting voltage, but this proved to be more difficult that in seemed. Mainly since this voltage level shifts based on what process needs to be undervolted. In general the simpler the process, the lower the voltage. Due to lack of time, we were unable to implement such a script reliably.

For the same reason, we did not include a function, which looks simple in concept, but isn't. We wanted to provide a way for the user to set a voltage (not the undervolting) via software means. We can do so with the micro-controller, but software undervolting requires the *undervoltage*, not the absolute voltage. We are convinced this could be done, but did not have the time to implement it. In our view it is a matter of writing to the right offset of MSR the right byte stream.

And lastly, there is a problem related to time in a different way. As has been stated several times, speed is absolutely critical to the correct performance of the library.

While using the convenience functions, many checks are performed to ensure the validity of the parameters. However, since we wanted the user to be able to take control over each step, we would have to include these checks at the beginning of *every public function*.

Furthermore, there is a potentially infinite number of checks we could do. We did try to include as many as possible in the above-discussed `plundervolt_faulty_undervolting_specification()` function |Safe Specification|, which is run before the undervolting starts, but afterwards, we faced a decision we were unable to resolve in a satisfying way.

If we included these checks, they would take too long to execute, while other threads/systems would not expect this delay. Conversely, if we omitted them, the library would be somewhat insecure, and prone to undefined behaviours.

As proof, consider the hardware example. Unlike with software undervolting, where we used global variables, we were forced to declare a structure with two integers, and multiply those together. Our computer did not undervolt when simple global or local variables were used, as they took less time to access, which threw the whole system off. Changing the execution times of the attack would have solved it, too, but you can see that such minute details make a huge difference.

As another example, imagine a simple software-based faulty multiplication attack. While a complicated check is performed, the other threads already run through the loops. The undervolting starts *after* the checks, and there isn't enough time to cause a fault before the multiplication loops end.

When it does end, a new voltage is set, lower than before, and the process repeats. In this way, the user would not achieve the faulty result, the library would lower the voltage on and on, past the actually faulting voltage, and either freeze the computer, or reach the `end_undervoltage` value, and be unsuccessful.

One such check we had to omit was the inclusion of the checking function pointer, which should point to the process that checks if the loops and undervolting should stop. There are several possible configurations, and they take too long to compute. Therefore, it may be possible (though hard, as all parameters are checked initially) to run a NULL function pointer instead of the checking function.

There are ways to fix this, and we have employed some of them. Every time a new configuration is set, the checks occur, for instance. Other fixes would have to be implemented by the user, such as a system of thread locks, and those would take execution time, too. In the end, we were forced to sacrifice security for efficiency.

5. Future work

In this section we are going to discuss some obvious or wanted extensions to the library. These are additions which we would have liked to do, but did not have time for.

The first thing that the users would appreciate is a printing system, which we could not get to work properly. Whether it is in a separate thread, or controlled by Teensy, the undervolting lasts only for a given period of time. While controlled by the user in many ways, such as the `wait_time` or various delay parameters, the library counts on a somewhat constant execution time.

This gets in the way of including a thorough debugging structure. The idea was to provide a way for the user to see what is happening down to the most miniscule operations through print statements. These would be optional, and ranging from the most general descriptions of what the library is doing (like "Creating threads") to printing the name of each function as it is called, with all its arguments, to printing a line for every iteration of every loop. Which level would be used would depend on a user-set global variable.

After implementing this, we found the library stopped working. Following some close examination we believe it is due to the fact that a print statement, a *conditional* print statement at that, and one executed in its own function (which would be the best course of actions, otherwise most of the library would be checks and print statements), takes A) too long; and B) not always the same amount of time.

We find the same issues with the first point - taking too long - as with security checks in each public function. The printing would take the time needed to undervolt, which causes the CPU to not make a mistake, and that leads to potential system freezes.

The second issue - non-constant time - is the real problem, though. Printing itself does not take long, but - since the level of printing and the number of printed lines depends on the user - the library cannot easily predict how long before the undervolting starts, and neither can the user. It is our opinion that such a system could be written, even though we failed to do so.

Another way to improve the library would be to include attack-specific functions. The function `plundervolt_run()` is a general convenience function, which executes whatever function it is told to, but the attack must still be written by the user.

We believed that we would have time to focus on one of the more complicated attacks (such as retrieving AES or RSA keys, manipulating with addresses and indices, etc.), and implement them, but this turned out to not be efficient - because it is not necessary, and therefore was not included in our crowded plan.

And finally, as is obvious, including more example usages to better showcase how to use the library with all its options would be prudent, but once again, we ran out of time.

6. Reflection

We are overall happy with the library as it is. We believe it is simple to use, and provides enough help to make it comfortable to do so.

Our initial goals were to create something friendly despite the complexities of the subject matter, and in this regard, we hope to have succeeded.

We have experienced unforeseen blocks along the way, which caused our major dissatisfactions with the library.

We felt neither of the original code was particularly well documented, so getting to understand what was happening took some time.

However, the greatest issue we faced was in continuous testing. As we have attempted to explain, the behaviour of an undervolted computer is somewhat unpredictable, in no small part due to changing circumstances. We estimate more than seventy per cent of our testing time was spent on figuring out whether an unexpected outcome was caused by our code, wrong parameters, or simply an unlucky try.

Unfortunately for us, this unexpected outcome usually resulted in a complete system crash, which meant a hard reset, and time wasted, most often only to freeze the computer again a short while later.

Because of this unpredictability, a lot more time was spent on fine tuning the library than we had planned, and so a lot of things we wanted to get to were simply left out in the end.

That being said, it is our belief the library will perform admirably, with all the desired flexibility and usefulness, and we are content with having made it.

There are many fixes and additions, usually small in nature, that we can think of right now. We are aware this is always going to be the case, of course, but in honestly judging our work, we would be putting forth an incomplete assessment had we not mentioned these ideas. Neither one, though, we are proud to say, should impede the usage of the library.

7. Conclusion

In this paper we have discussed the structure of our library, and the decisions which led to its final state. We have explained why voltage scaling attacks are dangerous, and why having a library to write them is beneficial.

Our goal was to implement a usable piece of code, available and helpful to programmers less knowledgeable of the undervolting process. We believe we have provided enough explanation of how the library can be used, what it does, and what its limitations are, to pronounce this goal successfully completed.

We have outlined the struggles we have faced during the library's creation, in order to better illustrate some of its functionalities and shortcomings. Moreover, we have justified our choices by describing how the library was required to perform, and by briefly examining the undervolting attacks.

We have included simple examples, and suggested modifications to them, to cover all possible (non-specific) usages of our library. In our view, this paper is enough to understand the concepts needed to test and explore undervolting, or create a new, similar attack.

Since we were heavily inspired by the works of the Plundervolt and VoltPillager papers, we focused in short on where our codes differ, and where (and why) we derived from them more closely.

To conclude, we have written a library, which we hope will prove valuable in further mending and improving the safety of of Intel's modern processors.

8. Acknowledgements

We would like to thank profoundly the authors of the Plundervolt and VoltPillager papers. Their work not only build the bases for this report, indeed formed most of the core principles in it, but proved to be great non-technical help during the creation of this paper. Namely, we thank Kit Murdock and Zitai Chen, who explained, and explained, and explained, with patience and (at least outward) understanding. Thank you.

9. References

1. Murdock K, Oswald D, Garcia FD, Van Bulck J, Gruss D, Piessens F. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: 2020 IEEE Symposium on Security and Privacy (SP). 2020. p. 1466–82.
2. Chen Z, Vasilakis G, Murdock K, Dean E, Oswald D, Garcia FD. VoltPillager: Hardware-based fault injection attacks against Intel {SGX} Enclaves using the {SVID} voltage scaling interface. In 2021 [cited 2021 May 5]. Available from: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai
3. Kenjar Z, Frassetto T, Gens D, Franz M, Sadeghi A-R. V0LTpwn: Attacking x86 Processor Integrity from Software. In 2020 [cited 2021 May 5]. p. 1445–61. Available from: https://www.usenix.org/conference/usenixsecurity20/presentation/kenjar
4. Qiu P, Wang D, Lyu Y, Qu G. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security [Internet]. New York, NY, USA: Association for Computing Machinery; 2019 [cited 2021 May 5]. p. 195–209. (CCS '19). Available from: https://doi.org/10.1145/3319535.3354201
5. Eleršič M. mihic/linux-intel-undervolt [Internet]. 2021 [cited 2021 May 5]. Available from: https://github.com/mihic/linux-intel-undervolt
6. Blömer J, Seifert J-P. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In: Wright RN, editor. Financial Cryptography. Berlin, Heidelberg: Springer; 2003. p. 162–81. (Lecture Notes in Computer Science).
7. Barenghi A, Bertoni G, Parrinello E, Pelosi G. Low Voltage Fault Attacks on the RSA Cryptosystem. In: 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). 2009. p. 23–31.
8. Kim Y, Daly R, Kim J, Fallin C, Lee JH, Lee D, et al. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. SIGARCH Comput Archit News. 2014 Jun 14;42(3):361–72.
9. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks: ACM SIGPLAN Notices: Vol 51, No 4 [Internet]. [cited 2021 May 5]. Available from: https://dl.acm.org/doi/abs/10.1145/2954679.2872390
10. Brasser F, Davi L, Gens D, Liebchen C, Sadeghi A-R. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In 2017 [cited 2021 May 5]. p. 117–30. Available from: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser
11. Costan V, Devadas S. Intel SGX Explained. IACR Cryptol ePrint Arch. 2016;2016(86):1–118.
12. van Schaik S, Kwong A, Genkin D, Yarom Y. SGAxe: How SGX Fails in Practice. :14.
13. Rotem E, Naveh A, Ananthakrishnan A, Weissmann E, Rajwan D. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. IEEE Micro. 2012 Mar;32(2):20–7.
14. RMClock Utility. Products. CPU Rightmark [Internet]. [cited 2021 May 5]. Available from: http://cpu.rightmark.org/products/rmclock.shtml
15. Tang BC. Security Engineering of Hardware-Software Interfaces [Internet]. Columbia University; 2018 [cited 2021 May 5]. Available from: https://doi.org/10.7916/D8CV5W7V
16. Badamasi YA. The working principle of an Arduino. In: 2014 11th International Conference on Electronics, Computer and Computation (ICECCO). 2014. p. 1–4.

10. Appendices

Appendix A: List of Public Functions

In this appendix, we will list all functions available to the user, with brief descriptions of their purpose. Fuller descriptions are provided in the library documentation.

We have split this list into three parts - functions used in every attack (general), functions used for software undervolting, and those used for hardware undervolting.

Subsection I: General Functions

- `plundervolt_faulty_undervolting_specification()`
  Checks if parameters are safe to use.
- `plundervolt_set_loop_finished()`
  Stops undervolting and all other loops run by the library, in all threads.
- `plundervolt_init()`
  Creates the parameter structure defined in the Setup section and in Appendix B. Returns this structure with default values.
- `plundervolt_set_specification()`
  Confirms user's choice of parameters. This is a necessary

step. Checks parameters are sensible.

- `plundervolt_apply_undervolting()`
  This function helps with preparing and running the attack. In case of software undervolting, it lowers the voltage while a function has to be running. When hardware-undervolting, it prepares the attack, and calls the function where the user sets the attack of.
- `plundervolt_run()`
  Performs all likely operations such as opening files, creating threads, calling `plundervolt_apply_undervolting()` and so on. It does not set specification, or close the files.
- `plundervolt_cleanup()`
  Closes all files to Teensy, the onboard trigger, and MSR.
- `plundervolt_error2str()`
  Returns a string description of an error passed as an argument.
- `plundervolt_print_error()`
  Prints the string returned by `plundervolt_error2str()` with newline.
- `plundervolt_reset_voltage()`
  After the attack, returns the system to its normal state, i.e., it returns the voltage to manageable levels, and resets the onboard pin. This is a necessary step.
- `plundervolt_open_file()`
  Opens the necessary files depending on the type of undervolting.
- `plundervolt_loop_is_running()`
  Returns `1` if there is a loop which repeatedly calls the undervolted function currently running.

Subsection II: Software-Undervolting Functions

- `plundervolt_compute_msr_value()`
  Computes the byte stream which will be written to MSR. This is necessary step when setting new voltage.
- `plundervolt_software_undervolt()`
  Sets new undervoltage, which is passed as an argument.
- `plundervolt_read_voltage()`
  The MSR can return the current voltage. This reads it, and returns it as `double`.
- `plundervolt_set_undervolting()`
  Sends instructions to MSR. Instruction are provided by `plundervolt_compute_msr_value()`. Is used by `plundervolt_software_undervolt()`.
- `plundervolt_get_current_undervoltage()`
  Returns the current undervoltage. Note, this is not the same as current voltage. Rather, it is the value in *mV* by which the current voltage differs from the base voltage for the current CPU frequency.

Subsection III: Hardware-Undervolting Functions

- `plundervolt_init_hardware-undervolting()`
  Prepares the micro-controller for the attack. This does not set up the attack, only the communication between the process and the micro-controller. Used by `plundervolt_open_file()`.
- `plundervolt_teensy_read_response()`
  Teensy sends responses as instructions are sent to it. The library reads them already, but if the user wishes to do so again, they can.
- `plundervolt_configure_glitch()`
  Set up the attack with library parameters.
- `plundervolt_arm_glitch()`
  Prepare Teensy to start the attack.
- `plundervolt_fire_glitch()`
  Start the hardware attack.

Appendix B: List of Undervolting Parameters

Subsection I: General Parameters

- `int loop`
  Values:
  - `0` … run the undervolted function only once.
  - `>0` … run the undervolted function in a loop, until `plundervolt_set_loop_finished()` is called.
  - special case: `>0` … in hardware undervolting, when `integrated_loop_check` is equal to 0, run as many times as this value, then stop.
- `void (* function)(void *)`
  A pointer to the undervolted function.
- `void * arguments`
  A *void* pointer to the arguments (usually as a user-defined structure) of the undervolted function.
- `int integrated_loop_check`
  Value >0 means the undervolted function calls `plundervolt_set_loop_finished()` on its own, so the library does not have to take care of it. When value is 0 or smaller, user must provide a function which stops the loops - the checking function.
- `int (* stop_loop)(void *)`
  A function pointer to the checking function.
- `void * loop_check_arguments`
  A *void* pointer to the arguments (usually as a user-defined structure) of the checking function.
- `int undervolt`
  Values:
  - `0` … Software undervolting is not performed, only the undervolted function is called. This is for testing purposes. It only has meaning for software undervolting, as hardware undervolting must be started by the user anyway (by calling `plundervolt_fire_glitch()`.
  - `>0` … run software undervolting in parallel to the undervolted function.
- `int wait_time`
  The number of *ms* the library sleeps for in strategically chosen places. This either gives the undervolting function time to start running, or (in case of software undervolting) dictates how long each voltage will be held for before it is lowered again.
  Note: This value only has meaning when using the convenience function `plundervolt_apply_undervolting()`.
- `undervolting_type u_type`
  Specifies which type of undervolting to perform. The enum type `undervolting_type` allows two options:
  - `software` … perform software undervolting.
  - `hardware` … perform hardware undervolting.

Subsection II: Software-Undervolting Parameters

- `uint64_t start_undervoltage`
  The beginning value of undervoltage (*not* voltage) for software undervolting. This is only relevant if using `plundervolt_apply_undervolting()`, as that function lowers the voltage gradually from this point.
- `uint64_t end_undervoltage`
  The ending value of undervoltage (*not* voltage) for software undervolting.
- `int threads`
  Values:
  - `0` … Only run the undervolted function in one thread.

    - `>0` … Specifies how many threads there are going to be for the undervolted function. Note that the value of 1 has the same effect as 0.
- `int step`
  The number of *mV* `plundervolt_apply_undervolting()` lowers the voltage by on each iteration.

Subsection III: Hardware-Undervolting Parameters

- `char* teensy_serial`
  The device connection to the micro-controller Teensy.
- `char* trigger_serial`
  The device connecting to the onboard trigger, which connects to Teensy. This is optional, but necessary if `using_dtr` is >0.
- `int teensy_baudrate`
  The speed of packets going to Teensy.
- `using_dtr`
  Values:
    - `0` … we only communicate with Teensy through the USB connection.
    - `>0` … we are using the onboard trigger (specified by `trigger_serial`) to start the attack.
- `int repeat`
  Once Teensy is told to start the attack, it can repeat the action. This parameter tells it how many times to do so.
- `int delay_before_undervolting`
  Teensy can wait after the attack starts before changing the voltage. This is how many *ms* it does so.
- `float start_voltage`
  The voltage at the beginning of the attack. Do not confuse with the software-focused `start_undervoltage`.
- `duration_start`
  How many *ms* `start_voltage` is held.
- `float undervolting_voltage`
  The voltage of the attack. The lowest voltage of the cycle, which should produce a fault.
- `int duration_during`
  How many *ms* `undervolting_voltage` is held.
- `end_voltage`
  The voltage which will be set at the end of the attack. Do not confuse with the software-focused `end_undervoltage`.
- `int tries`
  The convenience function `plundervolt_apply_undervolting()` will repeat the attack this many times.