Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86

Yingchen Wang* UT Austin
Riccardo Paccagnella* UIUC
Elizabeth Tang He UIUC
Hovav Shacham UT Austin
Christopher W. Fletcher UIUC
David Kohlbrenner UW

Abstract

Power side-channel attacks exploit data-dependent variations in a CPU's power consumption to leak secrets. In this paper, we show that on modern Intel (and AMD) x86 CPUs, power side-channel attacks can be *turned into* timing attacks that can be mounted without access to any power measurement interface. Our discovery is enabled by dynamic voltage and frequency scaling (DVFS). We find that, under certain circumstances, DVFS-induced variations in CPU frequency depend on the current power consumption (and hence, data) at the granularity of milliseconds. Making matters worse, these variations can be observed by a *remote attacker*, since frequency differences translate to wall time differences!

The frequency side channel is theoretically more powerful than the software side channels considered in cryptographic engineering practice today, but it is difficult to exploit because it has a coarse granularity. Yet, we show that this new channel is a real threat to the security of cryptographic software. First, we reverse engineer the dependency between data, power, and frequency on a modern x86 CPU—finding, among other things, that differences as seemingly minute as *a set bit's position in a word* can be distinguished through frequency changes. Second, we describe a novel chosen-ciphertext attack against (constant-time implementations of) SIKE, a post-quantum key encapsulation mechanism, that amplifies a single key-bit guess into many thousands of high- or low-power operations, allowing full key extraction via remote timing.

1 Introduction

Power-analysis attacks have been known for decades to be a powerful source of side channel information leakage. Historically, these attacks were used to leak cryptographic secrets from embedded devices like smart cards using physical probes [3,39,59,67,73,74]. Recently, however, power-analysis attacks have been shown to be exploitable also via software power measurement interfaces. Such interfaces, available on many of today's general-purpose processors, have been abused to fingerprint websites [94], recover RSA keys [69], break KASLR [63], and even recover AES-NI keys [64].

Fortunately, software-based power-analysis attacks can be mitigated and easily detected by blocking (or restricting [10]) access to power measurement interfaces. Up until today, such a mitigation strategy would effectively reduce the attack surface to physical power analysis, a significantly smaller threat in the context of modern general-purpose x86 processors.

In this paper, we show that, on modern Intel (and AMD) x86 CPUs, power-analysis attacks can be turned into timing attacks—effectively lifting the need for *any* power measurement interface. Our discovery is enabled by the aggressive dynamic voltage and frequency scaling (DVFS) of these CPUs. DVFS is a commonly-used technique that consists of dynamically adjusting CPU frequency to reduce power consumption (during low CPU loads) and to ensure that the system stays below power and thermal limits (during high CPU loads). We find that, under certain circumstances, DVFS-induced CPU frequency adjustments depend on the current power consumption at the granularity of milliseconds. Therefore, since the power consumption is data dependent, it follows transitively that CPU frequency adjustments are data dependent too.

Making matters worse, we show that data-dependent frequency adjustments can be observed without the need for any special privileges and even by a *remote* attacker. The reason is that CPU frequency differences directly translate to execution time differences (as 1 hertz = 1 cycle per second). The security implications of this finding are significant. For example, they fundamentally undermine constant-time programming, which has been the bedrock defense against timing attacks since their discovery in 1996 [58]. The premise behind constant-time programming is that by writing a program to only use "safe" instructions, whose latency is invariant to the data values, the program's execution time will be data-independent. With the frequency channel, however, timing becomes a function of *data*—even when only safe instructions are used.

Despite its theoretical power, it is not obvious how to construct practical exploits through the frequency side channel.

*These authors contributed equally to this work.

This is because DVFS updates depend on the aggregate power consumption over *millions* of CPU cycles and only reflect coarse-grained program behavior. Yet, we show that the frequency side channel is a real threat to the security of cryptographic software, by (i) reverse engineering a precise *leakage model* for this channel on modern x86 CPUs, and (ii) showing that some cryptographic primitives admit *amplification* of single key bit guesses into thousands of high- or low-power operations, enough to induce a measurable timing difference.

To construct a leakage model, we reverse engineer the dependency between data being computed on and power consumption / frequency on modern x86 Intel CPUs. Our results reveal that power consumption and CPU frequency depend on both the Hamming weight (HW) of data being processed and the Hamming distance (HD) of data across computations. We show, for the first time, that these two effects are distinct and additive on modern Intel CPUs. Further, the HW effect is non uniform. That is, computing on data with the same HW results in differences in power consumption / frequency depending on the *position* of individual 1s within data values. The takeaway is that computing on data with different bit patterns depending on a secret can result in different power consumptions and frequencies depending on that secret. We expect that this information will also be useful towards building future, Intel-specific power leakage emulators [11, 60, 71, 86, 88]. We find that AMD x86 CPUs also feature a similar leakage model, but leave reverse engineering its details to future work.

We then describe a novel attack, including new cryptanalytic techniques, on two production-ready, constant-time implementations of SIKE (Supersingular Isogeny Key Encapsulation [52]). SIKE is a decade old, widely studied key encapsulation mechanism. Unlike other finalists in NIST's Post-Quantum Cryptography competition, SIKE has both short ciphertexts and short public keys — and a "well-understood" side channel posture [20]. In our attack, we show that, when provided with a specially-crafted input, SIKE's decapsulation algorithm produces anomalous 0 values that depend on single bits of the key. Worse so, these values cause the algorithm to get *stuck* and operate on intermediate values that are also 0 for the remainder of the decapsulation. When this happens, the processor consumes less power and runs at a higher frequency than usual, and therefore decapsulation takes a shorter wall time. This timing signal is so robust that key extraction is possible across a network, as we demonstrate for the SIKE implementations in both Cloudflare's Interoperable Reusable Cryptographic Library (CIRCL) [28] and Microsoft's PQCrypto-SIDH [65]. Our unoptimized version of the attack recovers the full key from these libraries in 36 and 89 hours, respectively. Finally, we show that the frequency side channel can also be used to mount timing attacks without a timer, such as a KASLR break and a covert channel.

Disclosure We disclosed our findings, together with proof-of-concept code, to Intel, Cloudflare and Microsoft in Q3 2021 and to AMD in Q1 2022. The attack was assigned CVE-2022-23823 and CVE-2022-24436 and held under embargo until June 14, 2022. Intel committed to awarding us a bug bounty. Cloudflare and Microsoft deployed a mitigation to CIRCL and PQCrypto-SIDH, respectively.

2 Background and Related Work

Intel P-States In Intel processors, dynamic voltage and frequency scaling (DVFS) works at the granularity of P-states. P-states correspond to different operating points (voltage-frequency pairs) in 100 MHz frequency increments [49]. The number of P-states varies across different CPU models. Modern Intel processors offer two mechanisms to control P-states, namely SpeedStep and Speed Shift / Hardware Controlled Performance States (HWP). With SpeedStep, P-states are managed by the operating system (OS) using hardware coordination feedback registers. With HWP, P-states are managed entirely by the processor, increasing the overall responsiveness. HWP was introduced with the Skylake microarchitecture [77]. When HWP is enabled, the OS can only give hints to the processor's internal P-state selection logic, including restricting the range of available P-states [90]. Otherwise, the available range of P-states depends only on the number of active cores and on whether "Turbo Boost" is enabled [55]. Our P-state naming convention follows the one used in Linux [90].[1] The lowest P-state corresponds to the lowest supported CPU frequency. The highest P-state corresponds to the "max turbo" frequency for the processor. However, when Turbo Boost is disabled, the highest available P-state is the base frequency. We use the term P-state and frequency interchangeably.

P-state management is also related to power management. Each Intel processor has a Thermal Design Point (TDP), indicating the expected power consumption at steady state under a sustained workload [22, 40]. While in the max turbo mode, the processor can exceed its nominal TDP [47]. However, if the CPU hits a certain power and thermal limit while in max turbo mode, the hardware will automatically downclock the frequency to stay at TDP for the duration of the workload.

Data-Dependent Power Consumption It is well-known that a processor's power consumption depends on the data being processed [46, 67]. The precise dependency between data and power consumption depends on the processor's implementation, but can be approximated using leakage models. Two commonly-used leakage models are the Hamming distance (HD) [9, 61, 67, 70, 76] and the Hamming weight (HW) [56, 61, 66, 70, 72, 73, 87] models. In the HD model, power consumption depends to the number of $1 \to 0$ and $0 \to 1$ bit transitions occurring in the data during a computation. In the HW model, power consumption just depends on the number of bits that are 1 in the data being processed.

[1]However, Intel refers to higher frequencies as lower P-states [48, 50].

Table 1: CPUs tested in our experimental setups.

CPU Model	Microarchitecture	Cores	Base Frequency	Max Turbo Frequency
i7-8700	Coffee Lake	6	3.20 GHz	4.60 GHz
i7-9700	Coffee Lake Refresh	8	3.00 GHz	4.70 GHz
i9-10900K	Comet Lake	10	3.70 GHz	5.30 GHz
i7-11700	Rocket Lake	8	2.50 GHz	4.90 GHz
i7-10850H	Ice Lake (mobile)	6	2.70 GHz	5.10 GHz
i7-1185G7	Tiger Lake (mobile)	4	3.00 GHz	4.80 GHz

Power side-channel attacks Power side-channel attacks against cryptosystems were first publicly discussed by Kocher in 1998 [59]. His work introduced analytical techniques that exploit the data dependency of power consumption to reveal secret keys. Following works demonstrated power-analysis attacks against several cryptographic algorithms including AES [14, 66], DES [73], RSA [30, 74, 79, 93], and ElGamal [16,30].[2] However, all these attacks were targeted against smart cards and required physical access to the device. More recently, power side-channel attacks have been applied also to more complex devices such as smartphones [15,35,75,91,92] and PCs [36, 63, 64, 69, 94]. Some of these attacks rely only on software power measurement interfaces, meaning that they do not need proximity to the device. However, while some of these works use the HW and HD leakage models [64, 69], none of them presents a systematic reverse engineering of the dependency between power consumption and data on modern Intel x86 CPUs. Further, all these attacks can be blocked by restricting access to such power measurement interfaces.

3 CPU Frequency Leakage Channel

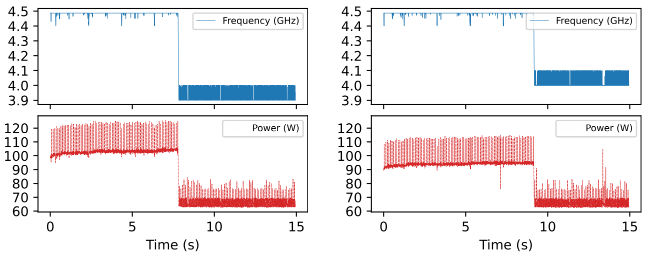
In this section, we analyze the leakage from CPU frequency variations on modern Intel processors. We show that, under certain circumstances, the distribution of a processor's frequencies leaks information about the instructions being executed as well as the data being processed.

Experimental Setup We run our experiments on several different machines. The characteristics of the CPU of each machine are reported in Table 1. All our machines run Ubuntu with versions either 18.04 or 20.04, kernel either 4.15 or 5.4, and the latest microcode patches installed. Unless otherwise noted, we use the default system configuration, without restricting the P-states. To monitor CPU frequency, we use the `MSR_IA32_MPERF` and `MSR_IA32_APERF` registers, as done in the Linux kernel [62]. To monitor power consumption, we use the MSRs of the RAPL interface, following Weaver [89].

3.1 Distinguishing Instructions

As a first step for our analysis, we set out to understand how running different workloads affects the P-state selection logic of our CPUs. We pick two workloads from the `stress-ng` benchmark suite [57]. The first workload consists of 32-bit integer and floating-point operations (`int32float` method), while the second workload consists of only 32-bit integer operations (`int32` method). We run both benchmarks on all cores and starting from an idle machine. We sample the CPU frequency and the (package domain) power consumption every 5 ms during the benchmark's execution.

[2]For a comprehensive survey of these attacks, we refer to prior work [67].



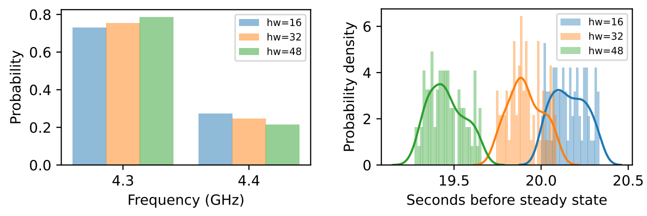
(a) Run of the `int32-float` test (b) Run of the `int32` test

Figure 1: Example of distinguishing workloads using frequency traces on our i7-9700 CPU. The lighter workload (`int32`) allows for longer runtimes at higher frequencies than the heavier workload (`int32-float`).

Figure 1a shows the results for the `int32float` test on our i7-9700 CPU. The frequency starts at 4.5 GHz, the highest P-state available when all cores are active on our CPU. This frequency is sustained for about 8 seconds, during which the power consumption is allowed to exceed the TDP. Then, the CPU drops to a lower P-state, bringing the power consumption down to TDP (65 W on our CPU). From there onwards, the CPU remains in *steady state* and power stays around the TDP level for the duration of the workload. In our example, at steady state the frequency oscillates between two P-states, corresponding to the frequencies of 3.9 GHz and 4.0 GHz.

Figure 1b shows the results for the `int32` stress test. Here too, the frequency starts at 4.5 GHz and later drops to a lower P-state. However, compared to Figure 1a, (i) the drop occurs later, after 10 seconds, and (ii) the P-states used after the drop are higher, corresponding to 4.0 GHz and 4.1 GHz. This is because the power consumption of the `int32` test is lower. As a consequence, not only can the processor sustain the highest available P-state for longer, but it can also use higher P-states in steady state without exceeding the TDP.

The key takeaway from the above results is that both (i) the time that a processor can spend at the maximum available P-state and (ii) the distribution of P-states at steady state depend on the CPU power consumption. Since the CPU power consumption depends on the workload, by the transitive property it follows that P-states depend on the workload too. This implies that *dynamic scaling of P-states leaks information about the current workload running on the processor.*



(a) Frequency at steady state (b) Seconds before steady state

Figure 2: Distinguishing data (in the source register to a `shlx` instruction) using frequency traces on our i7-9700 CPU. Figure 2a is over 30,000 samples. Figure 2b is over 100 traces.

3.2 Distinguishing Data

We saw that P-state information leaks information about the instructions being executed (i.e., the workload). We now explore if the frequency leakage channel can leak information about the data being processed by instructions. Our question is motivated by the fact that power consumption on x86 processors is known to be data dependent [64]. It is thus natural to ask: do data-dependent differences in power consumption show in the distribution of P-states?

To answer this question, we monitor the CPU frequency while executing the same instructions and only changing the content of the input registers. For example, we use the `shlx` instruction to continuously shift left the bits of a source register and write the result into different destination registers in a loop, while only varying the content of the source register. We run this experiment on all cores and compare the distribution of the P-states in steady state. Figure 2a shows the results when we set the content of the source register to have 16, 32 or 48 ones. In all cases the P-state oscillated between 4.3 GHz and 4.4 GHz. However, the larger the Hamming weight, the more the frequency stayed at the lower P-state. We also saw a data-dependent difference in terms of *when* the frequency drops to steady state if we start from idle (cf. Figure 2b). The larger the Hamming weight, the quicker the frequency drops to steady state. This is because, as we show in Section 4, processing data with larger Hamming weights consumes more power than processing data with lower Hamming weights.

We get similar results with other instructions too. For example, we observed data-dependent effects when running `or`, `xor`, `and`, `imul`, `add`, `sub`, as well as when computing on data loaded from memory. The only caveat is that, for some instructions, the power consumption of just running the target instruction in a loop on all cores was not large enough to cause the P-state to ever drop to steady state. In these cases, we ran an additional, fixed workload in the background to push the total power consumption up.

The key takeaway of the above results is that *dynamic scaling of P-states leaks information about the data being processed*. In the following sections, we use the distribution of P-states at steady state as our leakage channel.

4 CPU Frequency Leakage Model

We saw that the power consumption and the distribution of P-states in Intel CPUs depend on the data being processed. The goal of this section is to construct a leakage model of this behavior. To this end, we reverse engineer the dependency between power consumption/frequency and data on the ALU of modern Intel CPUs. As we show in Section 5, this information can help an attacker construct side-channel attacks.

Scope Precisely understanding where power is dissipated as a function of data on general-purpose x86 processors is a challenging task. The reason is that the microarchitecture of modern x86 processors is (i) highly complex and (ii) largely undocumented. Fortunately, studying the power consumption across all microarchitectural units is not necessary to build attacks. This is because the vast majority of computations performed by modern, constant-time cryptographic software occurs in the arithmetic logic unit (ALU). Since our primary goal is to build a model that is useful to leak secrets from constant-time cryptographic code, the analysis in this section focuses specifically on the ALU component.

Methodology We use the experimental setup of Section 3. In each experiment, we run a fixed set of ALU instructions (the *sender*) in a loop on all cores, while varying the input contents. We carefully design our senders to target specific behaviors and minimize side effects. First, to reduce power consumption from other core units such as the cache, we always use register to register instructions without any memory access. Second, to avoid any datapath contamination effects caused by incrementing the loop counter variable and evaluating loop conditions, we run our sender in an infinite loop that we manually terminate at the end of the experiment. Third, to avoid introducing unintended HD effects, we interleave different instructions in such a way that encourages full throughput on all available ports [1, 2]. Finally, we run each sender in two setups. In the first setup, we use the default system configuration, warm up the machine until it enters steady state, and monitor the frequency. In the second setup, we disable SpeedStep / HWP (ensuring that our machine always stays at the base frequency for the duration of the workload) and monitor the (core domain) power consumption. We sample power/frequency every 1 ms, collect 30,000 data points for each experiment and use their mean for our analyses.

4.1 Hamming Distance (HD) Effect

To start, we set out to understand if the number of $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions affects power consumption / frequency. Recall that these transitions depend on the number of bits that differ (also known as the HD) between consecutive data values being processed. To study the dependency between HD and power consumption / frequency, we then need a sender that offers fine-grained control over the number of transitions,

```
rax = COUNT
rbx = 0x0000FFFFFFFF0000
loop:
  shlx %rax,%rbx,%rcx    // rcx = rbx << rax
  shlx %rax,%rbx,%rdx    // rdx = rbx << rax
  shrx %rax,%rbx,%rsi    // rsi = rbx >> rax
  shrx %rax,%rbx,%rdi    // rdi = rbx >> rax
  shlx %rax,%rbx,%r8     // r8  = rbx << rax
  shlx %rax,%rbx,%r9     // r9  = rbx << rax
  shrx %rax,%rbx,%r10    // r10 = rbx >> rax
  shrx %rax,%rbx,%r11    // r11 = rbx >> rax
jmp loop
```

(a) Sender for our HD experiments.

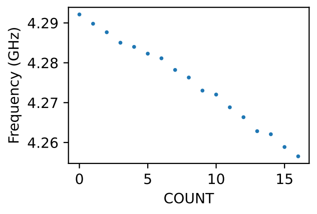
```
rax = LEFT
rcx = … = r11 = RIGHT
loop:
  or %rax,%rcx     // rcx = rax | rcx
  or %rax,%rdx     // rdx = rax | rdx
  or %rax,%rsi     // rsi = rax | rsi
  or %rax,%rdi     // rdi = rax | rdi
  or %rax,%r8      // r8  = rax | r8
  or %rax,%r9      // r9  = rax | r9
  or %rax,%r10     // r10 = rax | r10
  or %rax,%r11     // r11 = rax | r11
jmp loop
```

(b) Sender for our HW experiments.

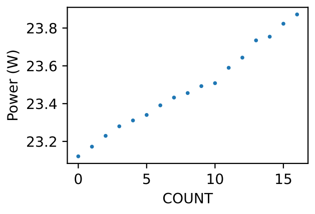
```
rax = rcx = rdx = rsi = rdi = FIRST
rbx = r8 = r9 = r10 = r11 = SECOND
loop:
  or %rax,%rcx     // rcx = rax | rcx
  or %rax,%rdx     // rdx = rax | rdx
  or %rax,%rsi     // rsi = rax | rsi
  or %rax,%rdi     // rdi = rax | rdi
  or %rbx,%r8      // r8  = rbx | r8
  or %rbx,%r9      // r9  = rbx | r9
  or %rbx,%r10     // r10 = rbx | r10
  or %rbx,%r11     // r11 = rbx | r11
jmp loop
```

(c) Sender for our HW+HD experiments.

Figure 3: Different sets of instructions (*senders*) used to reverse engineer the dependency between data and power consumption / frequency on our CPUs. Different senders are designed to target different effects. Each sender can be run with variable inputs.



(a) Frequency vs COUNT



(b) Power vs COUNT

Figure 4: Effect of increasing COUNT in Figure 3a's sender on our i7-9700 CPU. Higher COUNT values cause higher HDs in the ALU output. As the HD increases, the mean power consumption grows and the mean steady-state frequency drops.

while avoiding other potential side effects. For example, testing different HDs should not require changing the number of 1s in the input (which, as we show below, is a separate effect).[3]

We design our sender to use interleaved `shlx` and `shrx` instructions, as shown in Figure 3a. These instructions shift the bits of the second source register to the left or right by a COUNT value stored in the first source register. The result is written to a separate destination register. Since on our CPUs `shlx` and `shrx` execute on port 0 and port 6 [1], we interleave them in groups of two. We fix the content of the second source register to `0x0000ffffffff0000`, corresponding to 16 zeros, followed by 32 ones, followed by 16 zeros. We then shift this register left and right by COUNT (with $0 \leq$ COUNT ≤ 16).

By construction, the HD in the ALU output between a `shlx` and a `shrx` is $4 \times$ COUNT. For example, when COUNT $= 8$, the output of each `shlx` is `0x00ffffffff000000`, and the output of each `shrx` is `0x000000ffffffff00`, translating to 4×8 bit transitions in the ALU output. Yet, the ALU input remains unchanged and the number of 1s in the source and the destination registers is fixed.[4]

Figure 4 shows the results when we vary the COUNT value. We see that the power consumption grows and the frequency drops when COUNT grows, confirming that the number of bit transitions directly affects power consumption and frequency. In Appendix A.1, we corroborate this observation with an additional experiment where transitions occur in the ALU input. These results are consistent on all the CPUs of Table 1.

1. Larger Hamming distances between data values being processed contribute to larger power consumptions and lower steady-state frequencies.

4.2 Hamming Weight (HW) Effect

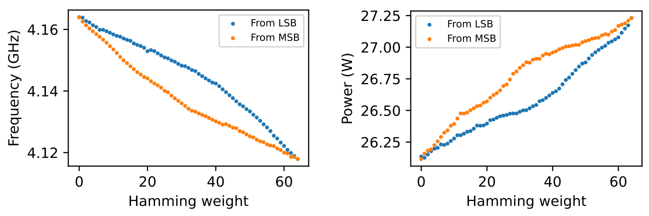
We now set out to understand if the HW of the data values being processed affects power consumption / frequency. Recall that the idea behind the HW model is that power consumption depends on the number of 1s in the data being processed. To study the dependency between HW and power consumption / frequency, we need a sender that offers fine-grained control over the number of 1s, while avoiding other potential side effects. For example, testing different HWs should not require bit transitions in the data (i.e., the HD effect).

To satisfy the above requirements, we design a sender that uses `or` logic instructions, as shown in Figure 3b. These instructions perform a bitwise inclusive or operation between the source register and the destination register, and store the result in the destination register. We always use the same input and output registers for all the `or` instructions in the loop. We fix the content of the source register to LEFT, and set the initial content of the output register to RIGHT.

By construction, the number of bit transitions occurring on the ALU input and output during the execution of the above sender is zero. The reason is that all `or` instructions take the same inputs and produce the same output during an experiment. Hence, we can test different HW in the source registers without introducing any HD effects. An added benefit of using `or` instructions is that they allow us to study the effects

[3]This requirement implies that approaches such as using a `xor` instruction to cause bit transitions are not suitable, because triggering different numbers of transitions would also require using different numbers of 1s in the input.

[4]The only other variable is the number of 1s in the COUNT register itself, which varies between 1 and 4. However, this effect is negligible.



(a) Frequency vs HW (b) Power vs HW

Figure 5: Effect of varying the number of consecutive 1s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. As we increase the number of 1s, the mean power consumption grows and the mean steady-state frequency drops.

of changing some bits of the input register (LEFT) without affecting the contents of the output register (RIGHT). We use this sender to perform multiple experiments.

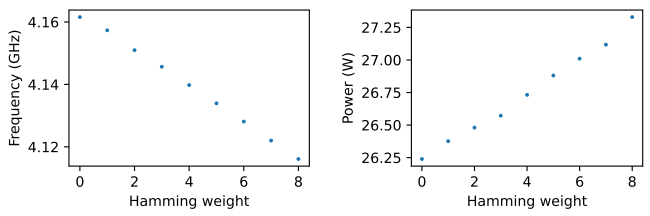
Consecutive 1s We start our analysis of the HW effect by checking if the number of leading or trailing 1s in the data affects power consumption / frequency. We set LEFT = RIGHT such that the inputs and outputs of all `or` instructions are always the same. We then run the sender with a varying HW in the LEFT = RIGHT values. Figure 5 shows the results when the HW grows from 0 to 64, both when the 1s start from the least significant bit (LSB) and when they start from the most significant bit (MSB). In both cases, the power consumption grows and the frequency drops when the HW grows.

2. A larger number of leading or trailing 1s in the data values being processed contributes to larger power consumptions and lower steady-state frequencies.

We also see that the changes in power consumption and frequency appear to be nonlinear. That is, the plots of Figure 5 have a "bow" shape, suggesting that the HW effect is stronger for the most significant 32 bits than for the least significant 32 bits. For example, when the input is `0xffffffff00000000` (HW=32, orange line), the HW effect is larger than when it is `0x00000000ffffffff` (HW=32, blue line). This suggests that given data values with the same HW, their contribution power / frequency may also depend on the position of 1s. We thoroughly examine this observation later in this subsection.

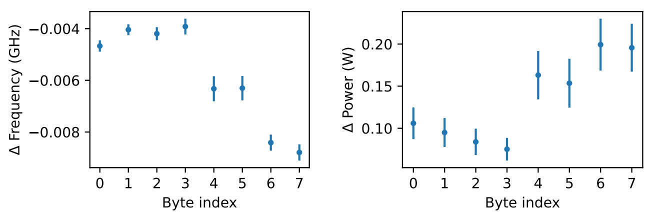
Non-consecutive 1s The above experiment shows that power consumption and frequency can depend on the HW of the data being processed. However, it only focuses on a bit pattern of consecutive 1s and 0s. In reality, 1s and 0s might occur in anywhere in the data. For our model to be useful, we need to test if the HW effect applies to arbitrary bit patterns.

To analyze the HW effect in the presence of non-consecutive 1s, we run a variant of our previous experiment, where we increase the HW at byte granularity. That is, we break the 64-bit registers LEFT = RIGHT into 8 bytes and



(a) Frequency vs HW (b) Power vs HW

Figure 6: Effect of varying the number of non-consecutive 1s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. The results confirm that larger HWs cause higher power consumptions and lower steady-state frequencies.



(a) Effect of 0xFF to frequency (b) Effect of 0xFF to power

Figure 7: Effect of setting single bytes to 0xff in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. The effect varies depending on the position of 1s within the inputs. HW differences in the MSBs have the strongest effect; HW differences in the bits right below 32 have the weakest effect.

vary the HW within each byte. Increasing the HW within each byte allows us to measure the impact of different numbers of non-consecutive 1s. For example, when the HW for each byte is 2, we set 2 bits of each byte to 1, for a total HW of $2 \times 8 = 16$. Figure 6 shows the results, clearly indicating that a larger number of non-consecutive 1s contributes to a larger power consumption and lower CPU frequency.

3. A larger Hamming weight (number of 1s) in the data values being processed contributes to larger power consumptions and lower steady-state frequencies regardless of whether the 1s are consecutive or not.

Non-uniformity of the HW effect To analyze the impact of the position of 1s within the data, we run another variant of our previous experiment. We break the 64-bit registers LEFT = RIGHT into 8 bytes. Each byte can be set to `0x00` (all 0s) or `0xff` (all 1s). When we target byte i, we fix the value of the other 7 bytes and compute the delta of power consumption / frequency between setting byte i to `0xff` and `0x00`. For each byte, we repeat this test with all the 2^7 combinations of the other 7 bytes. We compute the average and standard deviation of the deltas for each byte and show the result in Figure 7.

We immediately see that the HW effect is non-uniform across different bytes. At a high level, the 4 most significant bytes have a stronger HW effect than the 4 least significant bytes, and bytes closer to the 32nd bit have a weaker HW effect than bytes farther from the 32nd bit. This is consistent with our previous observation that an input where the most significant 32 bits are 1 consumes more power than an input where the least significant 32 bits are 1, even if their HWs are the same. Further, the standard deviations are relatively small, suggesting that the HW effect of each byte is independent of the values of other bytes. For example, the power/frequency deltas between `0x0000ff0000000000` and `0x0000000000000000` are the same as the ones between `0xff00ffff00ffffff` and `0xff0000ff00ffffff`. We suspect that these properties also hold a bit granularity, but are unable to confirm because it would require collecting data for 2^{64} bit combinations for a runtime of more than 10^{13} years. Note that the difference in the HW effect due to the position of 1s is relatively small (e.g., ≤ 0.12 W in Figure 7b) compared to the difference in the HW effect due to the number of 1s (e.g., ≤ 1.11 W in Figures 5b and 6b) and the HD effect due to bit transitions (e.g., ≤ 0.75 W in Figure 4b).

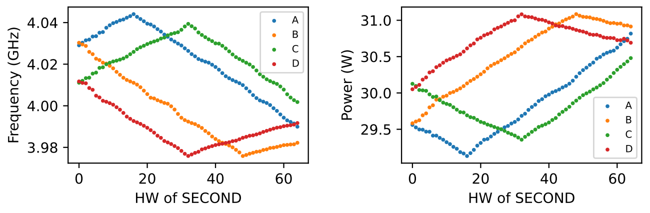
4. The HW effect is non-uniform. 1s in the most significant bytes affect power and frequency more than 1s in the least significant bytes. Additionally, the HW effect at each byte is independent of the values of other bytes.

The above experiments show that power consumption and frequency depend both on the number and the positions of 1s in the data being processed. However, both experiments were designed using LEFT = RIGHT, meaning that all the source and destination registers used by the sender during an experiment were the same. It is then natural to ask: does the HW effect occur even when LEFT $\neq$ RIGHT? To answer this question, we repeated the above two experiments, but this time set LEFT = 0 and only varied the HW of RIGHT.[5] Both experiments yielded results similar to the ones where LEFT = RIGHT, albeit with smaller increments/decrements in power/frequency. This result shows that the HW effect on an operand is independent of the contents of other operands.

5. The HW effect occurs on each operand independently.

To sum up, the HW effect may be approximated as a linear combination of two vectors. The first vector is the number of 1s per byte, and the second vector is the non-uniform power consumption / frequency "cost" of 1s in that byte (based on the deltas of Figure 7). In Appendix A.1 we discuss additional experiments in support of this model. We verified that this model applies to all the CPUs of Table 1. However, the non-uniform "costs" per byte of the HW effect can be different across CPU models. For example, in the 11th gen CPUs, the HW effect is more uniform compared to Figure 7.

[5]Whether LEFT = 0 or LEFT = RIGHT, the result of the or is still RIGHT.



(a) Frequency vs HW (b) Power vs HW

Figure 8: Effect of increasing the HW of SECOND in Figure 3c's sender, while fixing FIRST to different values on our i7-9700 CPU. Power consumption grows and steady-state frequency drops when both HW and HD increase at the same time (net effect of HW + HD). However, power consumption drops and steady-state frequency grows when HW increments correspond to HD decrements (net effect of HW − HD).

4.3 Additivity of the HW and HD Effects

Finally, we set out to understand if the HD and HW effects are additive. To this end, we design our sender to use `or` instructions with interleaved operand contents, as shown in Figure 3c. In this sender, half of the instructions computes FIRST|FIRST and the other half computes SECOND|SECOND. We interleave these instructions in groups of four, since on our CPUs `or` instructions use four ports [1]. We then test setting FIRST to be A = `0x000000000000ffff`, B = `0xffff000000000000`, C = `0x00000000ffffffff`, or D = `0xffffffff00000000`, and increase the HW of SECOND from 0 to 64, starting from the least significant bit.

Figure 8 shows the results. Consider the case when FIRST = C. As the HW of SECOND increases from 0 to 32, the HD between FIRST and SECOND decreases, causing the power consumption to drop and the frequency to grow. However, as HW of SECOND increases from 32 to 64, the HD between FIRST and SECOND increases, causing the opposite effect. The slope between 0 and 32 is smaller than the one between 32 and 64. This is because the former is a net effect of HW minus HD whereas the latter is a net effect of HD plus HW. For the other values of FIRST, we see analogous effects but with different constant offsets. This result (consistent across the CPUs of Table 1) shows that the HW and the HD effects can simultaneously contribute to power and frequency.

6. The HD and HW effects are additive and can simultaneously contribute to differences in power consumption and steady-state frequency.

5 Remote Timing Attack on SIKE

The previous sections have shown that carefully crafted instruction sequences can trigger data-dependent power consumption and frequency differences. In this section, we show that the frequency side channel threat extends to in-the-wild software. Specifically, we show how to use the frequency side channel, combined with novel cryptanalysis, for a full key recovery attack through *remote timing* on two production-ready, side-channel hardened implementations of Supersingular Isogeny Key Encapsulation (SIKE) [52], a post-quantum key encapsulation mechanism based on the Supersingular Isogeny Diffie-Hellman (SIDH) [53] key exchange protocol.

Attack Model We assume a chosen-ciphertext attack model (CCA). The goal of the attacker (client) is to recover the static secret key used by the victim (server) to decapsulate ciphertexts. The attacker can send many ciphertexts to the victim, which always tries to compute the shared secret with the decapsulation procedure using its static secret key.

Attack Idea The server's static secret key is an integer m with bit expansion $m = (m_{\ell-1}, \ldots, m_0)_2$, where $\ell = 378$ (for SIKE-751, the parameter selection we target in our experiments). During decapsulation, the server computes $P + [m]Q$ for elliptic curve points P and Q included in the ciphertext; the SIKE standard prescribes a particularly efficient algorithm for evaluating this expression, the Montgomery three-point ladder [29]. We show that an attacker who knows the i least significant bits of m can construct points P and Q such that:

- If $m_i \neq m_{i-1}$, then the $(i+1)$st round of the Montgomery three-point ladder produces an anomalous 0 value. Once that anomalous 0 value appears, the decapsulation algorithm gets *stuck*: every intermediate value produced for the remainder of the ladder is 0. Additionally, every intermediate value produced for the function (isogeny computation) following the ladder is also 0.

- If, however, $m_i = m_{i-1}$, or if the attacker was wrong about the i least significant bits of m when constructing the challenge ciphertext, then the $(i+1)$st round generates a non-0 value. Heuristically, the remainder of the computation proceeds without producing an anomalous 0 value except with negligible probability.

This observation is new, and it represents a core contribution of our work. Because SIKE is built on somewhat abstruse math, we defer the details of how to construct points P and Q that trigger an anomalous 0 value, and why a 0 value causes the decapsulation algorithm to get stuck, to Section 5.3.

The values operated on by SIKE decapsulation are large (a single element of the field underlying SIKE-751 takes 188 bytes to express) and the operations themselves are complex: the inner loop of the Montgomery ladder comprises thousands of lines of hand-optimized assembly. Nevertheless, in Section 5.1, we show that SIKE decapsulation behaves like the much simpler, synthetic senders of Section 4. When $m_i \neq m_{i-1}$ and the decapsulation algorithm gets stuck, repeatedly producing and operating on 0 values, the processor consumes less power and runs at a higher steady-state frequency (and therefore decapsulation takes a shorter wall time).

Taken together, our findings mean that the server's secret key can be recovered by an adaptive chosen-ciphertext attack, using execution time as a side channel. Having extracted the first i bits of m, the adversary repeatedly queries the server with ciphertexts that should cause decapsulation to get stuck in the $(i+1)$st round. If the server responds faster than a baseline (established through profiling), the adversary concludes that bit m_i is the opposite of bit m_{i-1}; otherwise bit m_i is the same. The attacker then proceeds to the next bit. In Section 5.2, we show that the timing signal is so robust that key extraction is possible *across a network*. We demonstrate full recovery of the (378-bit) private key from the SIKE-751 implementations in two popular, production-ready cryptographic libraries: Cloudflare's Interoperable Reusable Cryptographic Library (CIRCL) [28], written in Go, and Microsoft's PQCrypto-SIDH [65], written in C. Both of libraries are hardened against previously known software side channels and meant to run in constant time. Our attack is practical; an unoptimized version recovers the full key from a CIRCL server in 36 hours and from a PQCrypto-SIDH server in 89 hours.

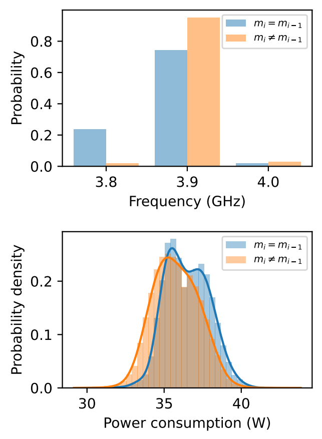
5.1 P-State and SIKE implementation

We start by verifying that a correct key-bit guess in our chosen-ciphertext attack—one that causes the Montgomery ladder and the remainder of SIKE decapsulation to repeatedly produce 0 values—causes the processor to execute at a higher frequency than an incorrect key-bit guess does. Our local experiment uses 10 randomly generated SIKE-751 server keys. For each key $m = (m_{\ell-1}, \ldots, m_0)_2$, we target 4 out of the 378 bit positions. We choose the target bit positions uniformly at random, to validate that the frequency difference is observable even for bits accessed late in the Montgomery ladder loop.

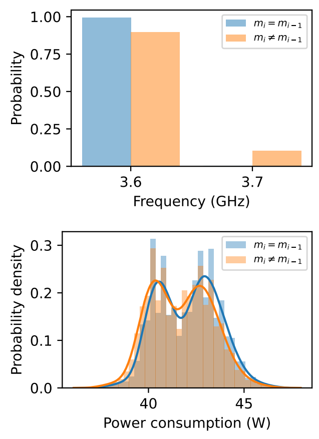
Suppose we target bit i in a secret key m. Provided that $m_i \neq m_{i-1}$, we can craft a challenge ciphertext that will trigger an anomalous 0 value in the Montgomery ladder iteration that accesses bit i. However, if $m_i = m_{i-1}$, then there is *no* challenge ciphertext that can trigger the anomalous 0 value. To make sure we are measuring the effect of anomalous 0 values, and not some other unknown effect, we set up our experiment as follows. For each key m and each target bit index i, we create a variant key m' that agrees with m at every bit position except index i, where it has the *opposite* bit value.[6] In other words, $m' = \big(m_{\ell-1}, \ldots, m_{i+1}, (1-m_i), m_{i-1}, \ldots, m_0\big)_2$. A challenge ciphertext crafted as described in Section 5.3.2 will induce an anomalous 0 against exactly one of m and m'.

For each key, m or m', and for each target bit position i, we launch a multithreaded SIKE decapsulation server. The server spawns 300 concurrent goroutines (CIRCL) or pthreads (PQCrypto-SIDH). Each thread handles a single decapsulation and then exits; when all threads have joined, we relaunch the server. We allow execution to continue until 800 seconds

[6]Every integer between 0 and $2^{378}-1$ is a valid SIKE-751 server private key. Given a private key we can compute the corresponding public key.



(a) CIRCL data



(b) PQCrypto-SIDH data

Figure 9: Distribution of the power consumption and the frequency when the challenge ciphertext introduces an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$), using the setups from Section 4 on our i7-9700 CPU. The results are over 10 randomly generated keys, where, for each key, we target 4 out of the 378 bit positions. For each key and each bit, we launch the server with 300 goroutines (CIRCL) or pthreads (PQCrypto-SIDH), each of which handles a single decapsulation request.

have elapsed. As in Section 4, we run each experiment in two setups. In the first setup, we use the default system configuration, and monitor the steady-state CPU frequency. In the second setup, we disable SpeedStep / HWP (ensuring that the frequency stays fixed at the base frequency) and monitor (core domain) power consumption. We sample both the CPU frequency and the power consumption every 1 ms.

We group the measured data points according to whether we expect the challenge ciphertext to induce an anomalous 0 or not. For each key m and target bit position i, exactly one of m and m' contributes to the anomalous-0 grouping.

The results, shown in Figures 9a and 9b, confirm that the steady-state frequency is higher and the power consumption is lower when an anomalous 0 is triggered ($m_i \neq m_{i-1}$) than when it is not ($m_i = m_{i-1}$), for both the CIRCL and the PQCrypto-SIDH decapsulation servers. As noted above, both these libraries are hardened against previously known software side channels and meant to run in constant time.

The signal we obtain from PQCrypto-SIDH is fainter than the one we obtain from CIRCL, because PQCrypto-SIDH uses a different strategy for Montgomery reduction that causes the value 0 to be represented in memory sometimes as 0 and sometimes as a prime number of size 751 bits.

5.2 SIKE Key Remote Recovery

We now show that the secret-dependent power consumption and frequency differences observed in Section 5.1 translate to a remotely observable secret-dependent *timing* difference.

We configure a SIKE target server with a randomly generated static 378-bit key for SIKE-751[7], revealed for comparison only after the attack completes. Our server accepts a client decapsulation request over HTTP (Go) or TCP (C) and spawns a goroutine (Go) or pthread (C) to handle the request. The thread reads in the ciphertext and performs the decapsulation computation, after which it sends a message back to the client indicating the establishment of a shared secret but no other information. The target server and the attacker are both connected to the same network, and we measure an average round-trip time of 688 µs between the two machines.

The attacker simultaneously sends n requests with a challenge ciphertext meant to trigger an anomalous 0 and measures the time t it takes to receive responses for all n requests. When an anomalous 0 is triggered, power decreases, frequency increases, SIKE decapsulation executes faster, and t should be smaller. Based on the observed t and the previously recovered secret key bits, the attacker can infer the value of the target bit, then repeat the attack for the next bit.

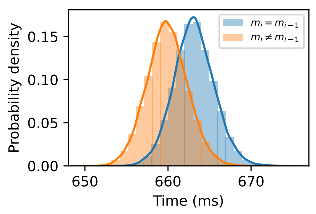
For the attack to be successful, we must overcome a number of practical difficulties. First, we must set a value for n, the number of requests, that allows us to observe a clear timing signal when we trigger the anomalous 0s. We experimentally find an n big enough that the frequency increase is remotely observable, but not so big that we induce thrashing.

Second, we must set a time cutoff to distinguish when anomalous 0s are triggered and when they are not. To this end, we collect the decapsulation times when querying the server with a random ciphertext, and use these times to set a cutoff for queries not triggering anomalous 0s. We then query the server with the challenge ciphertexts for the first few bits of the key until we see a speedup compared to the random ciphertext, and use these times to set a cutoff for queries triggering anomalous 0s.

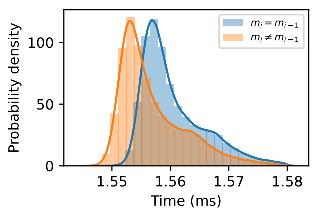
Third, we must detect and recover from mistakes caused by random variations in the server's decapsulation time. Recall that a challenge ciphertext constructed using a wrong value for the i least significant bits of m will never trigger anomalous 0s regardless of the relationship of m_i and m_{i-1}. Measuring no timing reduction in many consecutive rounds is evidence either that many consecutive key bits all have the same value (unlikely since key bits are independent and uniformly distributed), or that the value we are using for the least significant bits of the key is wrong (cf. Appendix A.4). In our experiments, we backtrack when experiments for 40 consecutive bit positions show no timing reduction.

Finally, there is a chance that a challenge ciphertext constructed as in Section 5.3.2 will accidentally trigger an anomalous 0 later in the decapsulation process even if it does not at the target bit index i of the Montgomery ladder. This will hap-

[7]The SIKE standard and the implementations we examined place the long-term keypair in the 3-torsion and the ephemeral key used for forming a ciphertext in the 2-torsion, so this is the case we studied. A variant of our attack applies also if the roles are swapped.



(a) CIRCL histogram



(b) PQCrypto-SIDH histogram

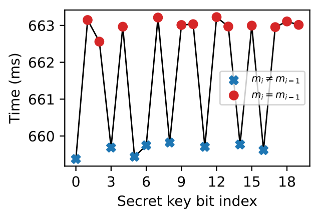
Figure 10: Distribution of the timings measured by the attacker during the remote key extraction attack, with the server running on an i7-9700 CPU. The attacker makes 300 (CIRCL) and 1000 (PQCrypto-SIDH) connections (all with the same challenge ciphertext, constructed as per Section 5.3.2) and measures the time until the last connection completes. We group the execution time (filtered) of each key bit extraction based on whether it should have triggered an anomalous 0 in the Montgomery ladder (i.e., whether $m_i = 1 - m_{i-1}$ or not).

pen with exponentially small probability for most bit indices, but larger probability for the last few bit indices. We defer a detailed explanation to Appendix A.3. It may be possible to avoid triggering this misbehavior with a different way of constructing the challenge key. We instead sidestep it by stopping our interaction with the server after extracting all but the last 14 bits; we recover these last bits by brute-force search.

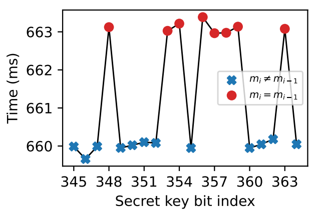
Attack Setup We run the SIKE target server on our i7-9700 CPU using the default system configuration. In the attack on CIRCL, the server is an HTTP server written using Go's `net.http` library, which handles each request in a goroutine. In the attack on PQCrypto-SIDH, the server is a TCP server written in C, which handles each request in a pthread.

We configure the attacker to send $n = 300$ concurrent requests in the CIRCL case, and $n = 1000$ requests in the PQCrypto-SIDH case. In both cases, concurrent requests are sent all with the same challenge ciphertext (constructed as described in Section 5.3.2), and the attacker measures the time until the last connection completes. We experimentally determine the expected timings when the CPU frequency increases because of anomalous 0s and when it does not: for CIRCL, at most 660.2 ms and at least 662.5 ms, respectively; for PQCrypto-SIDH at most 1556 ms and at least 1558 ms, respectively. We repeat the measurement 400 times, exclude outliers (CIRCL: below 650 ms or above 675 ms; PQCrypto-SIDH: below 1500 ms or above 1580 ms), compute the median of the remaining values, and compare to the cutoffs. If the result is inconclusive for a bit, we repeat the attack on that bit. We use our side channel to extract the key up to bit 364 and recover the last 14 bits by brute force search.

Results In Figure 10a and Figure 10b, we show the timing distribution of the 300-connection runs (CIRCL) and 1000-connection runs (PQCrypto-SIDH) respectively, grouped ac-

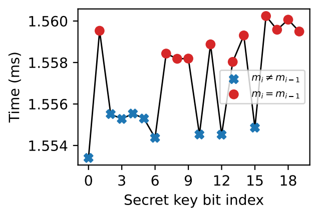


(a) CIRCL first 20 bits

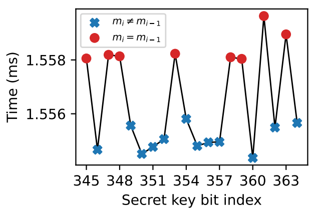


(b) CIRCL last 20 bits

Figure 11: Median times used to extract the first 20 bits (0 to 19) and the last 20 bits (345 to 364) of the key for the same attack against CIRCL SIKE-751 as in Figure 10a. The timings depend on whether the challenge ciphertext triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$).



(a) PQCrypto-SIDH first 20 bits



(b) PQCrypto-SIDH last 20 bits

Figure 12: Median times used to extract the first 20 bits (0 to 19) and the last 20 bits (345 to 364) of the key for the same attack against PQCrypto-SIDH SIKE-751 as in Figure 10b. The timings depend on whether the challenge ciphertext triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$).

cording to whether the challenge ciphertext of that run triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$).

For the first and the last 20 bit positions of the key that we extract by interacting with the server (bits 0–19 and 345–364, respectively), we plot, in Figure 11 (CIRCL) and Figure 12 (PQCrypto-SIDH), the median time among the 400 measurements for that bit and whether the run triggered an anomalous 0 ($m_i \neq m_{i-1}$) or not ($m_i = m_{i-1}$) at that bit position. The signal is strong for both the top bits and the bottom bits.

Both attacks successfully recovered the full secret key. The attack on CIRCL completed in 36 hours, while the attack on PQCrypto-SIDH completed in 89 hours. We expect that the attack running time could be reduced with careful optimization. Unlike our attack on CIRCL, our attack on PQCrypto-SIDH made 1 mistake and needed to backtrack; see Appendix A.4 for our error correction strategy.

5.3 Anomalous 0s in SIKE Decapsulation

We now explain how an attacker can construct SIKE ciphertexts that trigger an anomalous 0 in the $(i+1)$st iteration of the Montgomery ladder when $m_i \neq m_{i-1}$, and why that anomalous 0, once generated, causes the remainder of the decapsulation algorithm to also produce 0s repeatedly.

We briefly recall some relevant mathematical background in Appendix A.2. We recommend that readers review a longer introduction to the math behind elliptic curves, isogenies, and SIKE; Costello's tutorial expositions of elliptic curves [18] and isogenies [19] are especially good choices.

The first subroutine in the SIKE decapsulation algorithm recovers (the Montgomery coefficient A of) the curve E'_0 on which the points P, Q, and $Q-P$, included in the ciphertext provided by the attacker, lie. This subroutine is fast and independent of the secret key; we do not consider it further.

The second subroutine uses the Montgomery three-point ladder to compute $P+[m]Q$ on the curve E'_0 recovered by the first subroutine. This is the subroutine in which a correct key-bit guess ($m_i \neq m_{i-1}$) can trigger the generation of an anomalous 0 value. We explain how in Section 5.3.2.

The third subroutine evaluates the isogeny corresponding to the point $P+[m]Q$, computing (the Montgomery coefficient of) the curve E'_{e_3} that is the image of E'_0 under that isogeny. The fourth subroutine computes the j-invariant of the curve E'_{e_3}; this j-invariant is the shared SIDH secret. In Section 5.3.3 and Appendix A.3, we explain how an anomalous 0 value output by the Montgomery ladder causes the isogeny evaluation (third subroutine) and the j-invariant computation (fourth subroutine) to produce additional anomalous 0s.

The final step in SIKE decapsulation is a Fujisaki–Okamoto consistency check [31, 44] that checks that the ciphertext was properly generated. If the check fails, the recipient generates a random session key instead of the one prescribed by the (invalid) ciphertext. The Fujisaki–Okamoto check immunizes SIKE against attacks, such as that due to Galbraith et al. [32], that require partial information about the j-invariant computed when decapsulating (invalid) ciphertexts.

We do not claim to invalidate SIKE's proof of security. None of the ciphertexts we construct in our attack passes the Fujisaki–Okamoto check. Nevertheless, our attack recovers the server's secret key, because we obtain the information we need from the running time of the subroutines performed before the Fujisaki–Okamoto check.

While our paper was under embargo (cf. Section 1), our chosen-ciphertext attack triggering anomalous 0s in SIKE decapsulation, described in this subsection, was independently rediscovered by De Feo et al. [25].

5.3.1 Affine and Projective X-Coordinate Point Representations on Montgomery Curves

A Montgomery curve is defined by the equation $E_{A,B}\colon By^2 = x^3 + Ax^2 + x$, with parameters $A, B \in \mathbb{F}_{p^2}$ such that $B(A^2-4) \neq 0$. Montgomery curves have properties that make them suitable for efficient, side-channel resistant implementations. In particular, many operations needed in cryptography can be computed using just the x-coordinate of a point (ignoring the y-coordinate) and just the curve parameter A (ignoring the curve parameter B). The point with x- and y-coordinate both equal to 0 is a point of order 2 with special significance to arithmetic on a Montgomery curve; it is denoted by T.

Algorithm 1: Three point ladder ([52], Appendix A)

```
1 function Ladder3pt
      Input: $m = (m_{\ell-1}, \dots, m_0)_2 \in \mathbb{Z}$, $(x_P, x_Q, x_{Q-P})$,
             and $(A : 1)$
      Output: $(X_{P+[m]Q} : Z_{P+[m]Q})$
1 $((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)) \leftarrow ((x_Q : 1), (x_P : 1), (x_{Q-P} : 1))$
2 $a_{24}^{+} \leftarrow (A+2)/4$
3 for $i = 0$ to $\ell - 1$ do
4     if $m_i = 1$ then
5         $((X_0 : Z_0), (X_1 : Z_1)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), (a_{24}^{+} : 1))$
6     else
7         $((X_0 : Z_0), (X_2 : Z_2)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), (a_{24}^{+} : 1))$
8 return $(X_1 : Z_1)$
```

To minimize the need for (expensive) modular inversions, implementations typically work using projective rather than affine x-coordinate representation. For a point P, we write x_P for its affine x-coordinate and $(X_P : Z_P)$ for its projective x-coordinate representation, where $x_P = X_P \cdot Z_P^{-1}$. As usual, there are many equivalent $(X : Z)$ pairs that represent the same affine point. We write $(X : Z) \sim (X' : Z')$ to mean that there exists a scaling factor r such that $X = rX'$ and $Z = rZ'$.

The point T is represented as $(0 : 1)$ when using projective x-coordinates; the point at infinity, O, as $(1 : 0)$. The projective pair $(0 : 0)$ is *not* considered the valid projective x-coordinate representation of any point. This is important to our attack.

5.3.2 Anomalous 0s in the Montgomery Ladder

The Montgomery three-point ladder is implemented using `Ladder3pt` shown in Algorithm 1, reproduced from the SIKE specification [52]. The inputs are an integer m, curve points P, Q, and $Q-P$ (in affine x-coordinate representation), and the curve parameter A. The output is the point $P+[m]Q$ (in projective x-coordinate representation).

The subroutine invoked inside the loop, `xDBLADD`, when applied to points U, V, and $U-V$, returns a tuple consisting of $[2]U$ and $U+V$. As the names suggest, invoking `xDBLADD` is equivalent to invoking `xDBL` to compute $[2]U$ and `xADD` to compute $U+V$, but the combined algorithm evaluates some repeated subexpressions just once.

The points P, Q, and $Q-P$, as well as the curve parameter A, are supplied by the attacker, whereas the integer m is the secret key. The goal of the attacker is to leak m.[8]

[8] As written, algorithm `Ladder3pt` is not constant time, but the branch in line 4 is implemented in practice using constant-time conditional swaps.

Consider the algorithm xADD that, given points U, V, and W in projective x-coordinate form where $W = U - V$, computes the point $U + V$ in projective x-coordinate form, as:

$$\begin{aligned} X &\leftarrow Z_W \left[(X_U - Z_U)(X_V + Z_V) + (X_U + Z_U)(X_V - Z_V)\right]^2 \\ Z &\leftarrow X_W \left[(X_U - Z_U)(X_V + Z_V) - (X_U + Z_U)(X_V - Z_V)\right]^2 \ . \end{aligned}$$

When $U - V$ is any point except O or T, $\texttt{xADD}(U,V,U-V)$ correctly returns $U+V$. However, when $U - V$ is O or T, $\texttt{xADD}(U,V,U-V)$ misbehaves and returns the invalid projective representation $(0:0)$ instead of $U+V$ [21].[9]

Worse, $\texttt{xADD}(U,V,W)$ will also return $(0:0)$ if called with any of U, V, or W equal to $(0:0)$, regardless of the value of the other two inputs.[10] Repeated applications of xADD can thus get stuck at $(0:0)$. We use exactly this fact for our attack.

Suppose that we can arrange that, at the beginning of iteration k in Ladder3pt, $(X_2 : Z_2) \sim T$, i.e., that $X_2 = 0$ and Z_2 is nonzero. There are 2 cases to consider:

- if $m_k = 1$, then T will be passed into the third argument of xDBLADD, triggering the misbehavior in xADD and causing $(X_1 : Z_1)$ to be set to $(0:0)$.
- otherwise, if $m_k = 0$, then T will instead be passed into the *second* argument of xDBLADD. This will *not* trigger the misbehavior in xADD and not produce $(0:0)$ as an output. The point $(X_2 : Z_2)$, which was equal to T, will be overwritten with whatever xADD returns.

In the first case, xADD will get stuck; the second element of the tuple returned by xDBLADD will be $(0:0)$ in *every* subsequent iteration of Ladder3pt's loop, and Ladder3pt will eventually return $(0:0)$. In the second case, it is likely that 0 values will not recur during the ladder computation.

It remains to show how the attacker can arrange for $(X_2 : Z_2)$ to equal T at loop iteration k. Let $\mu_i = (m_{i-1}, \ldots, m_0)_2$ represent the least significant i bits of m. Algorithm Ladder3pt maintains the invariant that, at the beginning of iteration i of the loop, the points $(X_0 : Z_0)$, $(X_1 : Z_1)$, and $(X_2 : Z_2)$ satisfy

$$\begin{aligned} (X_0 : Z_0) &\sim [2^i]Q \\ (X_1 : Z_1) &\sim P + [\mu_i]Q \\ (X_2 : Z_2) &\sim (X_0 : Z_0) - (X_1 : Z_1) \ . \end{aligned}$$

Suppose that that the attacker, proceeding bit-by-bit, has extracted μ_k. The attacker picks an arbitrary curve and sets Q to be an arbitrary point on the curve.

If $m_{k-1} = 0$, the attacker sets

$$P \leftarrow \left[2^k - \mu_k\right] Q - T \ . \qquad (1)$$

Then, at iteration k of the Ladder3pt loop, we will have $(X_2 : Z_2) \sim T$. If $m_k = 1$, T will be passed as the third argument to xDBLADD, triggering the misbehavior as described above.

If $m_{k-1} = 1$, the attacker instead sets

$$P \leftarrow T - \left[\mu_k\right] Q \ . \qquad (2)$$

Then, at iteration k of the Ladder3pt loop, we will have $(X_1 : Z_1) \sim T$. If $m_k = 0$, T will be passed as the third argument to xDBLADD, triggering the misbehavior.

To summarize, if $m_k \neq m_{k-1}$, the crafted input ciphertext will trigger the anomalous 0 misbehavior.

When generated according to the SIKE specification, P and Q are always linearly independent points of order 3^{e_3} and never produce T or O during the execution of Ladder3pt. When generated according to our algorithm above but with an incorrect key-bit guess, we expect that T or O will be produced only with negligible probability.[11] This conjecture is supported by our experiments.

5.3.3 Anomalous 0s in Isogeny Evaluation and j-Invariant Calculation

The next task in SIKE decapsulation, isogeny evaluation, is carried out by algorithm 3_e_iso, which takes as input the point $P + [m]Q$ (in projective x-coordinate form) as output by Ladder3pt, expecting it to be a point of exact order 3^{e_3}. In Appendix A.3, we show that, when invoked on the invalid input $(0:0)$, 3_e_iso and its subroutines repeatedly operate on and produce 0 values. Isogeny evaluation in 3_e_iso thus acts as an amplifier for the signal produced by the ladder evaluation in Ladder3pt, making it possible to observe even an anomalous 0 produced in a late Ladder3pt loop iteration.

After isogeny evaluation, the next task in SIKE decapsulation is j-invariant calculation, using algorithm jInvariant. When 3_e_iso returns $(0:0)$, jInvariant is invoked with input $(0:0)$, every intermediate value it computes is 0, and its return value (the SIDH shared secret) is 0.[12]

5.4 Mitigations

We now describe the mitigation that Cloudflare and Microsoft deployed after we disclosed our attack on SIKE.

The mitigation, which was originally proposed by De Feo et al. [25], consists of validating that the ciphertext (public key) consists of a pair of linearly independent points of the correct order 3^{e_3}. This check is performed before running the three-point ladder and prevents attack ciphertexts from being further processed, thus hindering the attack. When running decapsulation on a single thread on our i7-9700 CPU, we

[9] If $U - V = O$ then $U = V$ and therefore $(X_U : Z_U) \sim (X_V : Z_V)$. If $U - V = T$ then $U = \tau_T(V)$ where τ_T is the translation-by-T map; by a property of Montgomery curves, it follows that $(X_U : Z_U) \sim (Z_V : X_V)$.

[10] In this case it does not matter — indeed, does not make sense to ask — whether $W = U - V$.

[11] This fact allows us not only to distinguish a correct from an incorrect bit guess for bit m_k but also to detect and recover from mistakes in determining the earlier bits μ_k; see Appendix A.4.

[12] Note that this output depends on the result of inverting 0 in $\mathbb{F}_{p^2}$ in step 15 of jInvariant. The Montgomery inversion algorithms in the implementations we examined have $1/0 = 0$ (see Savas and Koç [82]).

found that the mitigation adds a performance overhead of 5% for CIRCL and of 11% for PQCrypto-SIDH.

6 Timer-free Attacks

We now show that not only can we use the frequency side channel to turn power attacks into remote timing attacks (as we saw in Section 5), but we can also use it to mount timing attacks without a timer. To this end, we use the frequency side channel to mount a KASLR break and a covert channel.

KASLR Break Like prior work [12,13,37,43,45,51,63,64], the goal of the (unprivileged) attacker is to de-randomize the kernel base address. Knowledge of the kernel base address is useful to mount memory corruption exploits.

In Linux, the kernel text is placed at a 2 MB boundary in the `0xffffffff80000000` – `0xffffffffc0000000` range [13]. Hence, the kernel can be placed at one of 512 possible offsets. Prior work has shown that, on Intel and AMD processors, there is a timing and power consumption difference when executing prefetch instructions on a memory address depending on whether that address is mapped or not [43, 63]. This difference can be used to infer the location of the kernel within its predefined region. We show that this power consumption difference manifests also as a CPU frequency difference.

To this end, we build a sender process similar to the ones of Figure 3, but using only `prefetcht0` instructions. While the sender runs, a separate thread measures the current CPU frequency using the unprivileged `scaling_cur_freq` interface from the `cpufreq` driver. We ran the sender with all the 512 possible kernel base addresses, for 10 different randomizations (i.e., repeating across 10 reboots) on our Intel i7-9700 CPU. In all 10 cases, we were able to identify the base address successfully (as verified by checking the privileged `/proc/kallsyms` interface). We measured an average steady-state CPU frequency of 4.04 GHz when repeatedly prefetching mapped addresses, and 4.24 GHz when repeatedly prefetching unmapped addresses. The runtime of our unoptimized, proof-of-concept implementation is of 2 minutes. This runtime is larger than state-of-the-art KASLR breaks, but could be reduced with additional engineering effort.

Covert Channel Like prior work, our covert channel uses a *sender* and a *receiver*. To transmit a 0, the sender executes a loop of `or` instructions with high HD and HW in their data flow. This loop increases the power consumption and results in lower CPU frequency values. To transmit a 1, the sender executes a loop of `shlx` instructions with low HD and HW in their data flow. This loop decreases the power consumption and results in higher CPU frequency values. The receiver measures the current CPU frequency using the unprivileged `scaling_cur_freq` interface from the `cpufreq` driver.

We evaluated our covert channel by transmitting 1 kB of random data on our i7-9700 CPU. Our unoptimized, proof-of-concept implementation achieved a bandwidth of 30 bps with an error rate of 0.03% (average across 10 runs). This bandwidth is similar to the one of prior covert channels relying on software-based power measurement interfaces [63, 64].

7 Discussion

Affected CPUs We successfully reproduced our attack on Intel CPUs from the 8th to the 11th generation of the Core microarchitecture (reported in Table 1). We also tested two desktop CPUs from older generations, namely the i7-6700K (Skylake) and i7-7700K (Kaby Lake), and we found that both models only support Turbo frequencies on single core workloads: as soon as more than 1 core is active, the P-state is capped at the base frequency. In our experiments, we were not able to force the frequency into steady state (i.e., below the max turbo frequency) with single-core workloads, and were therefore unable to reproduce our attack on these models.

Besides CPUs from the (client-class) Core microarchitecture, our attack should also work on Intel Xeon CPUs (server-class), since they also use similar P-state management techniques. Additionally, other CPU vendors implement similar DVFS mechanisms and are likely vulnerable. For example, we verified that the AMD Ryzen processors are also vulnerable to our attack, featuring a similar HW/HD leakage model and enabling the same SIKE vulnerability that we described in Section 5. We leave reverse engineering the specific characteristics of the AMD leakage model to future work.

Closing the Frequency Channel In the affected CPUs, the root cause of our attack is DVFS in the Turbo frequency range. Intel CPUs use either one of two mechanisms to control DVFS: SpeedStep and HWP (cf. Section 2). To block leakage through the frequency channel, a defender may then decide to either disable Turbo Boost, or to disable both SpeedStep and HWP from the BIOS. We verified that both methods cause the P-state to always stay at the CPU base frequency for the duration of the workload, trading off performance for security. Alternatively, leakage might be mitigated by making DVFS in the Turbo frequency range less aggressive in future hardware.

Eliminating Leakage in Ciphers Another line of defense consists of removing secret-dependent leakage in cryptographic software. For example, introducing ciphertext validity checks before the Montgomery ladder evaluation in SIKE, as discussed in Section 5.4, prevents our attack by avoiding the long sequence of computations on zeros.

For cryptographic software in general, mitigating the power leakage itself would naturally close the frequency channel. True decoupling would require that all operands have no statistical correlation with secrets, which is only feasible with techniques like fully homomorphic encryption. A more realistic approach takes advantage of the fact that it is not the power usage of *each* operand that is leaked, but an average of the power usage across all operands in a time period. This goal may be achieved using masking/blinding techniques. Prior works have introduced protocol-specific masking techniques for ciphers such as AES [8,38,81,85] and blinding techniques for elliptic curve cryptography [54]. Automatic masking techniques have also been proposed either in software [7,17,27] or leveraging additional hardware support [26,33,41,42,78]. However, masked/blinded implementations may still leak in practice via power side channels [4,5,34,68,80,83,84].

Future defenses could also examine the potential of fusing unrelated loops, vectorizing operations, or other methods of interleaving different computations. These approaches could be done by combining multiple, normally sequential, computations in the program or by introducing an additional complementary kernel. Effective blinding will require that the combined computation power trace is no longer directly related to any secret computation. As an example, if we can construct a complementary (bit-inverted) version of a cryptographic kernel, we can interleave the real kernel and the blinding kernel. A detailed power model is essential to any of these approaches. Our model of HW and HD provides a strong starting point for any future work on blinding.

8 Conclusion

We discovered that in modern Intel (and AMD) x86 CPUs, DVFS-induced frequency variations depend on the current power consumption, and hence on the data being processed. We showed, for the first time, that the HD and HW of data individually and non-uniformly contribute to power consumption and frequency on modern x86 CPUs. We described a novel chosen-ciphertext attack against SIKE, which uses this knowledge to leak full cryptographic keys via remote timing.

The security implications of our findings are significant. Not only do they expand the attack surface of power side-channel attacks by removing the need for power measurement interfaces, but they also show that, even when implemented as constant time, cryptographic code can still leak via remote timing analysis. The takeaway is that current cryptographic engineering practices for how to write constant-time code are no longer sufficient to guarantee constant time execution of software on modern, variable-frequency processors.

Acknowledgments

This work was funded in part through NSF grants 1942888 and 1954521, and gifts from Google, Mozilla, and Qualcomm. Wang was partly supported by a Packard Fellowship (via Brent Waters). We thank our shepherd Michael Schwarz and the anonymous reviewers for their valuable feedback.

Availability

We have open sourced the code of all the experiments of this paper at https://github.com/FPSG-UIUC/hertzbleed.

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. In *ASPLOS*, 2019.

[2] Andreas Abel and Jan Reineke. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *ICS*, 2022.

[3] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, 3rd Edition*. John Wiley & Sons, 2020.

[4] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *CARDIS*, 2014.

[5] Sven Bauer. Attacking exponent blinding in RSA without CRT. In *COSADE*, 2012.

[6] Daniel J. Bernstein and Tanja Lange. Montgomery curves and the Montgomery ladder. In *Topics in Computational Number Theory Inspired by Peter L. Montgomery*. Cambridge University Press, 2017.

[7] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. Optimal first-order boolean masking for embedded iot devices. In *CARDIS*, 2017.

[8] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of AES. In *SAC*, 2004.

[9] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, 2004.

[10] Len Brown. powercap: restrict energy meter to root access. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=949dd0104c496fa7c14991a23c03c62e44637e71, 2020. Accessed on Jun 7, 2022.

[11] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. SoK: Design tools for side-channel-aware implementions. In *ASIACCS*, 2022.

[12] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, 2019.

[13] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break it, fix it, repeat. In *ASIACCS*, 2020.

[14] Suresh Chari, Charanjit Jutla, Josyula R Rao, and Pankaj Rohatgi. A cautionary note regarding evaluation of AES candidates on smart-cards. In *AES2*, 1999.

[15] Yimin Chen, Xiaocong Jin, Jingchao Sun, Rui Zhang, and Yanchao Zhang. POWERFUL: Mobile app fingerprinting via power analysis. In *INFOCOM*, 2017.

[16] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES*, 1999.

[17] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *FSE*, 2015.

[18] Craig Costello. Pairings for beginners. Online: https://www.craigcostello.com.au/s/PairingsForBeginners.pdf, 2012.

[19] Craig Costello. Supersingular isogeny key exchange for beginners. In *SAC*, 2019.

[20] Craig Costello. The case for SIKE: A decade of the supersingular isogeny problem. Cryptology ePrint Archive, Report 2021/543, 2021.

[21] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic - the case of large characteristic fields. *J. Cryptogr. Eng.*, 8(3), 2018.

[22] Ian Cutress. Why Intel processors draw more power than expected: TDP and Turbo explained. `https://www.anandtech.com/show/13544/why-intel-processors-draw-more-power-than-expected-tdp-turbo`, 2018. Accessed on Jun 7, 2022.

[23] Luca De Feo. Mathematics of isogeny based cryptography. Preprint, arXiv:1711.04062 [cs.CR], 2017.

[24] Luca De Feo. Exploring isogeny graphs. Habilitation thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2018.

[25] Luca De Feo, Nadia El Mrabet, Aymeric Genêt, Novak Kaluđerović, Natacha Linard de Guertechin, Simon Pontié, and Élise Tasso. SIKE channels. Cryptology ePrint Archive, Report 2022/054, 2022.

[26] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting RISC-V against side-channel attacks. In *DAC*. IEEE, 2019.

[27] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *CAV*, 2014.

[28] Armando Faz-Hernández and Kris Kwiatkowski. *Introducing CIRCL: An Advanced Cryptographic Library*. Cloudflare, 2019. `https://github.com/cloudflare/circl`. Accessed on Jun 7, 2022.

[29] Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11), 2018.

[30] Pierre-Alain Fouque and Frédéric Valette. The doubling attack–why upwards is better than downwards. In *CHES*, 2003.

[31] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1), 2013.

[32] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In *ASIACRYPT*, 2016.

[33] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. Cryptology ePrint Archive, Report 2020/773, 2020.

[34] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or foe? *TCHES*, 2020.

[35] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *CCS*, 2016.

[36] Daniel Genkin, Itamar Pipman, and Eran Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. In *CHES*, 2014.

[37] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *CCS*, 2020.

[38] Jovan D Golić and Christophe Tymen. Multiplicative masking and power analysis of AES. In *CHES*, 2002.

[39] Louis Goubin and Jacques Patarin. DES and differential power analysis the "duplication" method. In *CHES*, 1999.

[40] Corey Gough, Ian Steiner, and Winston Saunders. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, 2015.

[41] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. Concealing secrets in embedded processors designs. In *CARDIS*, 2016.

[42] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *TIS*, 2016.

[43] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*, 2016.

[44] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In *TCC*, 2017.

[45] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 2013.

[46] Intel. Running average power limit energy reporting / cve-2020-8694 , cve-2020-8695 / intel-sa-00389. `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html`. Accessed on Jun 7, 2021.

[47] Intel. Thermal design power (TDP) in Intel processors. `https://www.intel.com/content/www/us/en/support/articles/000055611/processors.html`. Accessed on Jun 7, 2022.

[48] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, June 2021.

[49] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2021.

[50] Intel. Power management - technology overview. `https://builders.intel.com/docs/networkbuilders/power-management-technology-overview-technology-guide.pdf`, 2021. Accessed on Jun 7, 2022.

[51] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *CCS*, 2016.

[52] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE. Technical report, National Institute of Standards and Technology, 2020.

[53] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *PQCrypto*, 2011.

[54] Marc Joye and Christophe Tymen. Protections against differential analysis for elliptic curve cryptography. In *CHES*, 2001.

[55] Manuel Kalmbach, Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. TurboCC: A practical frequency-based covert channel with Intel Turbo Boost. Preprint, arXiv:2007.07046 [cs.CR], 2020.

[56] Nikolaos Kavvadias, Periklis Neofotistos, Spiridon Nikolaidis, CA Kosmatopoulos, and Theodore Laopoulos. Measurements analysis of the software-related power consumption in microprocessors. *IEEE Transactions on Instrumentation and Measurement*, 53(4), 2004.

[57] Colin Ian King. stress-ng. `https://github.com/ColinIanKing/stress-ng`, 2022. Accessed on Jun 7, 2022.

[58] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

[59] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, 1999.

[60] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In *COSADE*, 2018.

[61] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An accurate instruction-level energy consumption model for embedded RISC processors. *ACM SIGPLAN Notices*, 36(8), 2001.

[62] Linux. `aperfmperf.c`. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/kernel/cpu/aperfmperf.c`. Accessed on Jun 7, 2022.

[63] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD prefetch attacks through power and time. In *USENIX Security*, 2022.

[64] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based power side-channel attacks on x86. In *S&P*, 2021.

[65] Patrick Longa. *Post-quantum Cryptography*. Microsoft, 2019. Available at https://github.com/microsoft/PQCrypto-SIDH. Accessed on Jun 7, 2022.

[66] Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In *ICISC*, 2002.

[67] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, volume 31. Springer Science & Business Media, 2008.

[68] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked AES hardware implementations. In *CHES*, 2005.

[69] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. How secure is green IT? the case of software-based energy side channels. In *ESORICS*, 2018.

[70] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *CHES*, 2000.

[71] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering:'grey box'modelling for instruction leakages. In *USENIX Security*, 2017.

[72] Thomas Messerges. Using second-order power analysis to attack DPA resistant software. In *CHES*, 2000.

[73] Thomas Messerges, Ezzy Dabbish, and Robert Sloan. Investigations of power analysis attacks on smartcards. In *USENIX Smartcard*, 1999.

[74] Thomas Messerges, Ezzy Dabbish, and Robert Sloan. Power analysis attacks of modular exponentiation in smartcards. In *CHES*, 1999.

[75] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. PowerSpy: Location tracking using mobile device power analysis. In *USENIX Security*, 2015.

[76] Jeremy Morse, Steve Kerrison, and Kerstin Eder. On the limitations of analyzing worst-case dynamic energy of processing. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3):1–22, 2018.

[77] Hassan Mujtaba. [IDF15]Intel's 6th gen Skylake unwrapped - CPU microarchitecture, Gen9 graphics core and Speed Shift hardware P-state. https://wccftech.com/idf15-intel-skylake-analysis-cpu-gpu-microarchitecture-ddr4-memory-impact/4/, 2015. Accessed on Jun 7, 2022.

[78] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *ICICS*, 2006.

[79] Roman Novak. SPA-based adaptive chosen-ciphertext attack on RSA implementation. In *PKC*, 2002.

[80] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *COSADE*, 2017.

[81] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, 2010.

[82] Erkay Savas and Çetin Kaya Koç. Montgomery inversion. *J. Cryptogr. Eng.*, 8(3), 2018.

[83] Werner Schindler and Andreas Wiemers. Power attacks in the presence of exponent blinding. *J. Cryptogr. Eng.*, 4(4), 2014.

[84] Werner Schindler and Andreas Wiemers. Generic power attacks on RSA with CRT and exponent blinding: new results. *J. Cryptogr. Eng.*, 7(4), 2017.

[85] Kai Schramm and Christof Paar. Higher order masking of the AES. In *CT-RSA*, 2006.

[86] Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. *NDSS*, 2021.

[87] Ankush Varma, Eric Debes, Igor Kozintsev, and Bruce Jacob. Instruction-level power dissipation in the Intel XScale embedded microprocessor. In *Embedded Processors for Multimedia and Communications II*, 2005.

[88] Nikita Veshchikov. SILK: high level of abstraction leakage simulator for side channel analysis. In *PPREW*, 2014.

[89] Vince Weaver. Reading RAPL energy measurements from linux. http://web.eece.maine.edu/~vweaver/projects/rapl/. Accessed on Jun 7, 2022.

[90] Rafael J. Wysocki. `intel_pstate` CPU performance scaling driver. https://www.kernel.org/doc/html/v4.19/admin-guide/pm/intel_pstate.html. Accessed on Jun 7, 2022.

[91] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A study on power side channels on mobile devices. In *Internetware*, 2015.

[92] Qing Yang, Paolo Gasti, Gang Zhou, Aydin Farajidavar, and Kiran Balagani. On inferring browsing activity on smartphones via USB power analysis side-channel. *IEEE Trans. Inf. Forensics Secur.*, 12(5), 2016.

[93] Sung-Ming Yen, Wei-Chih Lien, SangJae Moon, and JaeCheol Ha. Power analysis by exploiting chosen message and internal collisions - vulnerability of checking mechanism for RSA-decryption. In *Mycrypt*, 2005.

[94] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. Red alert for power leakage: Exploiting Intel RAPL-induced side channels. In *ASIACCS*, 2021.

A Appendix

A.1 Leakage Model—Additional Experiments

HD in the ALU Input In Section 4.1, we saw that increasing the number of bit transitions in the ALU output causes an increase in power consumption and a decrease in frequency. Here, we set out to understand if the same effect happens when bit transitions occur in the ALU input. We need a sender that offers fine-grained control over the number of transitions in the ALU input, while avoiding potential side-effects such as the HW effect or bit transitions in the ALU output.

We design a sender that is symmetric to the one of Figure 3a. Our sender still uses `shlx` and `shrx` instructions, as shown in Figure 13a. However, it is designed such that the output of all `shlx` and `shrx` instructions is always the same, and only their input varies as a function of COUNT. Hence, any HD effect is caused by bit transitions on the ALU input only. For example, when COUNT $= 8$, the source register to each `shlx` contains `0x000000ffffffff00`, and the source register to each `shrx` contains `0x00ffffffff000000`, the alternation of which translates to a HD of 4×8 in the ALU input.

Figure 14 shows the results for increasing COUNT values. We see that the power consumption grows and the frequency drops when COUNT grows, confirming that the number of bit transitions (i.e., the HD) in the ALU input directly affects power consumption and CPU frequency. We also see that the changes in power / frequency become more significant when the COUNT > 8, as a result of the non-uniform HW cost of having 1s closer to the MSB in the fixed source register `rcx`.

Non-uniform HW In Section 4.2, we saw that the HW effect it depends on the position of 1s in the data (i.e., it is non-uniform). We now discuss two experiments that provide additional evidence that the HW effect is non-uniform. We

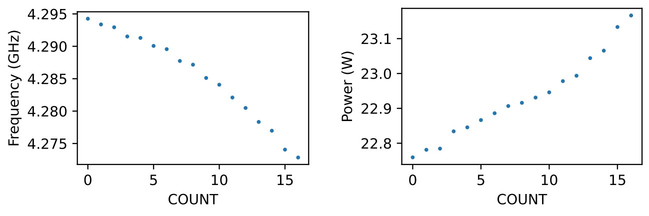
```
rax = COUNT
rbx = 0x0000FFFFFFFF0000 >> COUNT
rcx = 0x0000FFFFFFFF0000 << COUNT
loop:
  shlx %rax,%rbx,%rdx  // rdx = rbx << rax
  shlx %rax,%rbx,%rsi  // rsi = rbx << rax
  shrx %rax,%rcx,%rdi  // rdi = rcx >> rax
  shrx %rax,%rcx,%r8   // r8  = rcx >> rax
  shlx %rax,%rbx,%r9   // r9  = rbx << rax
  shlx %rax,%rbx,%r10  // r10 = rbx << rax
  shrx %rax,%rcx,%r11  // r11 = rcx >> rax
  shrx %rax,%rcx,%r12  // r12 = rcx >> rax
jmp loop
```

(a) Variant of sender for the HD experiments.

```
rax = 1
rsp = pointer_to_memory
rbx = … = r15 = INPUT
loop:
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
  mov %rax,(%rsp)  // store rax to memory
jmp loop
```

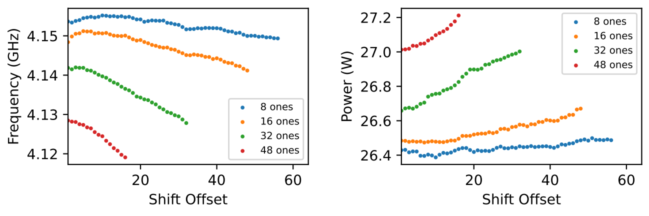
(b) Sender for the HW at rest experiments.

Figure 13: Additional sets of instructions (*senders*) used to reverse engineer the dependency between data and power consumption / frequency on our CPUs. Different senders are designed to target different effects. Each sender can be run with variable inputs.



(a) Mean frequencies. (b) Mean power consumptions.

Figure 14: Effect of increasing COUNT in Figure 13a's sender on our i7-9700 CPU. Higher COUNT values cause higher HDs in the ALU output. As the HD increases, the mean power consumption grows and the mean steady-state frequency drops.



(a) Frequency vs shift offset (b) Power vs shift offset

Figure 15: Effect of shifting consecutive 1s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. As we shift the 1s towards the MSB, the mean power consumption grows and the mean steady-state frequency drops.

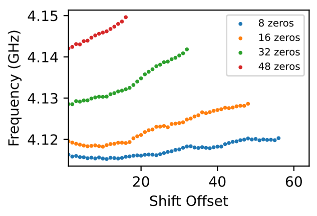
refer to these experiments as *shift*$_0$ and *shift*$_1$. Both experiments use the same sender of Section 4.2, shown in Figure 3b. In *shift*$_1$, we fix the number of consecutive 1s and measure the impact of changing the position of these consecutive 1s in the LEFT = RIGHT input, when all surrounding bits are 0s. In *shift*$_0$, we do the opposite: we fix the number of consecutive 0s and measure the impact of changing the position of these consecutive 0s in the LEFT = RIGHT input, when all surrounding bits are 1s. By construction, since the HW is fixed and the sender does not introduce any HD effect, any differences in the results depend only on the position of 1s.

We label different positions of the consecutive bit patterns based on their "shift offset" starting from the LSB. For example, when the number of consecutive 1s in *shift*$_1$ is 32, a shift offset of 0 refers to input value `0x00000000ffffffff` and a shift offset of 16 refers to input value `0x0000ffffffff0000`. Similarly, when the number of consecutive 0s in *shift*$_0$ is 32, a shift offset of 16 refers to input value `0xffff00000000ffff`.

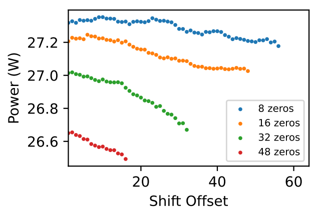
Figure 15 shows the results for the *shift*$_1$ experiment when we fix the number of consecutive 1s to 8, 16, 32, or 48. Consider the case when the number of 1s is 16. When the shift offset is between 0 to 16, we see almost no variation in the mean power consumption and steady-state frequency. This is because as we shift in this range, 1s are still all the low 32 bits, and we know from Figure 7 that there is little difference in the HW effect for 1s that are in the low 32 bits. However, when the shift offset increases from 16 to 48, the power consumption grows and the frequency drops. This is because we start gaining 1s in the high 32 bits and approaching the MSB. This behavior is consistent with what we saw in Figure 7, where 1s closer to the MSB have a stronger HW effect than 1s closer to the 32nd bit. The results are similar when the number of 1s is 8. When the number of 1s is 32 or 48, the HW effect increases every time the shift offset increases. The reason is that, in these cases, shifting means that we lose 1s in the low 32 bits and gain 1s in the high 32 bits, and we know from Figure 7 that 1s in the high 32 bits have a stronger HW effect than 1s in the low 32 bits. The HW increments in these cases are also more significant, because the delta between the HW effect of the bits we gain and the bits we lose is larger.

Figure 16 shows the results for the *shift*$_0$ experiment. These results are symmetrical to the *shift*$_1$ ones and can be explained by the same reasons described for the *shift*$_1$ experiment.

In summary, the *shift*$_0$ and *shift*$_1$ experiments support our observation that the HW effect is non-uniform.

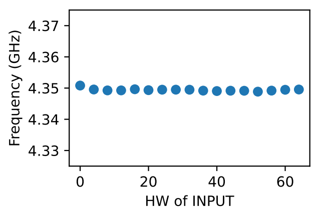


(a) Frequency vs shift offset

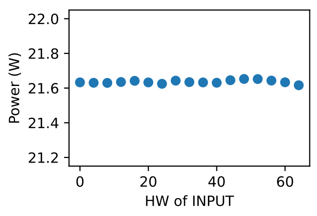


(b) Power vs shift offset

Figure 16: Effect of shifting consecutive 0s in the LEFT = RIGHT input to Figure 3b's sender on our i7-9700 CPU. As we shift the 0s towards the MSB, the mean power consumption drops and the mean steady-state frequency grows.



(a) Frequency vs HW



(b) Power vs HW

Figure 17: Effect of increasing the HW of INPUT (at rest) in Figure 13b's sender on our i7-9700 CPU. As we increase HW from 0 to 64, the mean power consumption and the mean steady-state frequency do not change.

HW Root Cause In Section 4.3, we saw that the HD effect and the HW effect are additive. Recall that the HD effect is due to $1 \to 0$ and $0 \to 1$ bit transitions in the data being processed. This is a well-understood effect in the literature, and can be attributed to the fact that when more bits flip during a computation, more transistors are switched in the datapath, which causes dynamic power consumption to grow [46, 67]. However, it is difficult to pinpoint the root cause of the HW effect on x86 Intel CPUs. For example, it is unclear if the HW effect occurs only when data is actively computed on, or if it is due to any data-dependent power cost of simply keeping data stored inside registers. Our sender from Figure 3b cannot distinguish between these two cases because it is designed to continuously compute on and overwrite identical data values. Here, we design a new sender to test if the HW effect occurs also when data values with different HWs are simply stored into registers (at rest), but not actively computed on.

Our sender, shown in Figure 13b, is designed as follows. First, it sets the content of `rax` to 1, `rsp` to a memory location, and all other architectural registers to a fixed INPUT value. Then, it enters an infinite loop of stores that write the content of `rax` into the memory location pointed to by `rsp`.[13]

By construction, the store operations in the loop are always the same and independent of the value of INPUT. Changing the value of INPUT only affects the content of registers that are initialized, but not actively computed on by the sender. Any difference in power consumption due to different INPUT values would then be due to HW effect at rest.

Figure 17 shows the results when we increase the HW of INPUT from 0 to 64. We see no differences in the mean power consumption and mean steady-state frequency when the HW grows. This result suggests that the HW effect does not occur when simply keeping data stored inside registers.

A.2 Mathematical Preliminaries for SIKE

SIKE is an isogeny-based key encapsulation method which involves arithmetic operations of elliptic curves over finite fields. In particular, SIKE uses Montgomery elliptic curves. Its security relies on the hardness of finding a specific isogeny between two such elliptic curves. Here, we provide an overview of the details of SIKE that are relevant to our attack.[14]

Let p be a prime of the form $2^{e_2}3^{e_3} - 1$. SIKE works in the field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 = -1 \pmod{p}$ and uses the supersingular elliptic curves over $\mathbb{F}_{p^2}$ that have $(2^{e_2}3^{e_3})^2$ points. The set of points $P \in E(\overline{\mathbb{F}_p})$ that satisfy $[n]P = O$ is called the n-torsion of E. The curves of interest were chosen so that the entire $(2^{e_2}3^{e_3})$-torsion is already defined over $\mathbb{F}_{p^2}$, and we have $E[2^{e_2}3^{e_3}] \cong \mathbb{Z}/(2^{e_2}3^{e_3})\mathbb{Z} \times \mathbb{Z}/(2^{e_2}3^{e_3})\mathbb{Z}$; as a result, for each curve of interest, $E[2^{e_2}]$ can be generated by linear combinations of two points P_2 and Q_2 with coefficients in $\mathbb{F}_{p^2}$; and likewise $E[3^{e_3}]$ can be generated by linear combinations of two points P_3 and Q_3 with coefficients in $\mathbb{F}_{p^2}$.

An isogeny $\phi\colon E_1(\mathbb{F}_{p^2}) \to E_2(\mathbb{F}_{p^2})$ is a group homomorphism from $E_1(\mathbb{F}_{p^2})$ to $E_2(\mathbb{F}_{p^2})$ and a non-constant rational map defined over $\mathbb{F}_{p^2}$ that preserves the point at infinity O. The kernel of an isogeny is $\ker\phi = \{P \in E_1 : \phi(P) = O\}$.

Every finite subgroup H of a curve $E(\mathbb{F}_{p^2})$ defines an isogeny $\phi\colon E \to E/H$, unique up to isomorphism, such that $\ker\phi = H$. The cardinality of H is also the degree of the rational map ϕ. Given H, Vélu's algorithm allows the rational map for the isogeny corresponding to H to be computed; the computation is tractable when $|H|$ is small.

An ℓ-isogeny is defined as $\phi_\ell : E \to E/\langle P \rangle$, where P has exact order ℓ. The order of $\phi(Q)$ in $E/\langle P \rangle$ is the same as the order of Q in E unless Q lies above $\ker\phi$ (meaning that $[n]Q \in \ker\phi \setminus \{O\}$ for some n), in which case the order of $\phi(Q)$ is reduced by a factor of ℓ.

Two curves are isogenous (meaning that there exists an isogeny from one to the other) if they have the same number of points. The curves of interest are all isogenous. If there is an isogeny ϕ from E_1 to E_2 then there is also an inverse

[13]We use a store so that the register file is constantly being read from, in the offchance an inactive register file could be powered down.

[14]For more information on SIKE, we refer to the SIKE tutorial by Costello [19] and to the SIKE specification [52]. For more information on elliptic curves and isogenies, we refer to the pairings tutorial by Costello [18] and to De Feo's lecture notes [23] and habilitation thesis [24]. For more information on Montgomery ladders, we refer to Bernstein and Lange [6].

Algorithm 2: Computing and evaluating a 3^e-isogeny, simple version ([52], Appendix A)

```
1 function 3_e_iso
    Static parameters: Integer $e_3$ from the public parameters
    Input: Constants $(A_{24}^+ : A_{24}^-)$ corresponding to a curve $E_{A/C}$, $(X_S : Z_S)$ where $S$ has exact order $3^{e_3}$ on $E_{A/C}$
    Output: $(A_{24}^{+\prime} : A_{24}^{-\prime})$ coresponding to the curve $E_{A'/C'} = E/\langle S \rangle$
1 for $e = e_3 - 1$ downto 0 by $-1$ do
2     $(X_T : Z_T) \leftarrow$ xTPLe$((X_S : Z_S), (A_{24}^+ : A_{24}^-), e)$
3     $((A_{24}^+ : A_{24}^-), (K_1, K_2)) \leftarrow$ 3_iso_curve$((X_T : Z_T))$
4     if $e \neq 0$ then
5         $(X_S : Z_S) \leftarrow$ 3_iso_eval$((K_1, K_2), (X_S : Z_S))$
6 return $(A_{24}^+ : A_{24}^-)$
```

isogeny ϕ^{-1} from E_2 to E_1; the composition $\phi^{-1} \circ \phi$ is the map $[\deg \phi]$ on E_1, and likewise $\phi \circ \phi^{-1}$ on E_2.

If isogenies ϕ and ϕ^{-1} between E_1 and E_2 are such that $\phi^{-1} \circ \phi$ is the identity map [1] then E_1 and E_2 are isomorphic. Isomorphic curves share the same j-invariant; the j-invariant of a curve can be computed given a description of the curve.

A.3 Anomalous 0s in Isogeny Evaluation: Details

Algorithm 2 reproduces the implementation of 3_e_iso from the specification [52] with optional arguments omitted.[15]

3_e_iso calls xTPLe to compute $[3^e](X_S : Z_S)$, by repeated application of the tripling map xTPL; if $(X_S : Z_S)$ has exact order 3^{e_3} then $(X_T : Z_T)$ is a point of exact order 3. 3_iso_curve expects a point of exact order 3 as input and uses Vélu's algorithm to compute the isogeny corresponding to the group $\langle (X_T : Z_T) \rangle$. It also outputs the curve constant for the curve that is the image of E under that isogeny. Finally, 3_iso_eval uses the values returned by 3_iso_curve to compute the image of $(X_S : Z_S)$ under the isogeny. Because $(X_S : Z_S)$ lies above $(X_T : Z_T)$, which is in the kernel of the isogeny, the order of the image of $(X_S : Z_S)$ is 3^{e-1}. We refer to the SIKE specification [52] for a more detailed description of the xTPLe, 3_iso_curve and 3_iso_eval algorithms.

When the attacker has made a correct key-bit guess using the methods of Section 5.3.2, Ladder3pt, which is supposed to return $P + [sk_3]Q$ in projective x-coordinate form, instead returns the invalid value $(0 : 0)$. This is not a valid projective representation of any point on E_0', and certainly not the representation of a point of exact order 3^{e_3}.

[15]In fact, optimized implementations evaluate the isogeny using a more complicated but more efficient strategy. Our attack applies to either approach.

An examination of the subroutines invoked by 3_e_iso reveals some remarkable facts:

- If xTPL or xTPLe is called with $(0 : 0)$ as its first argument, every intermediate value it computes is 0, and its return value is $(0 : 0)$, regardless of its second argument.
- If 3_iso_curve is called with $(0 : 0)$ as its argument, every intermediate value it computes is 0, and it returns $(0 : 0)$ for $(A_{24}^+ : A_{24}^-)$ and $(0, 0)$ for (K_1, K_2).
- If 3_iso_eval is called with $(0 : 0)$ as its first argument, every intermediate value it computes is 0, and its return value is $(0 : 0)$, regardless of the value of its second argument (which, in this case, is $(0, 0)$).

As a result, when 3_e_iso is invoked on input $(X_S : Z_S) = (0 : 0)$, every single intermediate value computed during every loop iteration of 3_e_iso is a 0, including every intermediate value computed in every subroutine that 3_e_iso calls.

For inputs generated according to our algorithm in Section 5.3.2 above but with an incorrect key-bit guess, the point $(X_S : Z_S)$ on which 3_e_iso is called will not be $(0 : 0)$, but it will also not be a point of exact order 3^{e_3}, so the behavior of 3_e_iso on it is unspecified. On such inputs, we expect that 3_e_iso will get stuck on $(0 : 0)$ only with negligible probability. This conjecture is likewise supported by our experiments, with one caveat noted above: there is some chance that the challenge ciphertext we form to target a bit accessed late in the ladder will cause Ladder3pt to output a point of order 3^e for some $e \lneq e_3$.[16] Evaluating 3_e_iso on such a point will trigger isogenies computed with kernel as the group formed by point of infinity O, which will trigger a frequency increase. We sidestep this problem by recovering the last 14 bits of the key by brute-force search.

A.4 SIKE Error correction

During our attack, a mistake made at bit position k invalidates measurements targeting all subsequent bit positions. With m_k correct, we expect to observe frequency increases (and thus $m_k \neq m_{k-1}$) with probability $1/2$. By contrast, with m_k incorrect we expect to *never* to observe frequency increases.

After a sufficiently long run without observing frequency increases, we backtrack to find the misinterpreted bit. In Section 5's experiments, we set the backtrack threshold to 40 bits.

[16]In our challenge ciphertext, we set $Q = P_2 + P_3$, so Q is a point of order $2^{e_2}3^{e_3}$. The output of Ladder3pt is a linear combination of T and $[2^i]Q$, $[a]T + [b][2^i]Q$, where a and b depend on the secret m. When i is large, $[2^i]P_2$ will have small order $2^{e_2 - i}$. Since T is a point of order 2, $[a]T$ and $[b][2^i]P_2$ might happen to cancel each other, and $[a]T + [b][2^i]Q$ will end up as $[b][2^i][P_3]$. If b happens to be a multiple of 3, the output of Ladder3pt would be a point of order smaller than 3^{e_3}.