DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data

Yingchen Wang*, Riccardo Paccagnella†, Alan Wandke†, Zhao Gang*, Grant Garrett-Grossman†, Christopher W. Fletcher†, David Kohlbrenner‡, Hovav Shacham*

*University of Texas at Austin, †University of Illinois Urbana-Champaign, ‡University of Washington

***Abstract*—The recent Hertzbleed disclosure demonstrates how remote-timing analysis can reveal secret information previously only accessible to local-power analysis. At worst, this constitutes a fundamental break in the constant-time programming principles and the many deployed programs that rely on them. But all hope is not lost. Hertzbleed relies on a coarse-grained, noisy channel that is difficult to exploit. Indeed, the Hertzbleed paper required a bespoke cryptanalysis to attack a specific cryptosystem (SIKE). Thus, it remains unclear if Hertzbleed represents a threat to the broader security ecosystem.**

In this paper, we demonstrate that Hertzbleed's effects are wide ranging, not only affecting cryptosystems beyond SIKE, but also programs beyond cryptography, and even computations occurring outside the CPU cores. First, we demonstrate how latent gadgets in *other* cryptosystem implementations—specifically "constant-time" ECDSA and Classic McEliece—can be combined with *existing cryptanalysis* to bootstrap Hertzbleed attacks on those cryptosystems. Second, we demonstrate how power consumption on the integrated GPU influences frequency on the CPU—and how this can be used to perform the first cross-origin pixel stealing attacks leveraging "constant-time" SVG filters on Google Chrome.

1. Introduction

Hertzbleed is a recently-discovered class of timing attacks that can leak secrets even from correctly implemented constant-time code [1]. Unlike traditional timing attacks, which are only able to leak "digital features" (e.g., instruction count) related to a program's execution, Hertzbleed can turn power side channels into timing attacks. As a result, Hertzbleed can leak "analog features" (e.g., features of instruction operands) that could not be previously leaked via timing attacks. This finding, made possible by the ubiquitous dynamic voltage and frequency scaling feature of modern hardware, has challenged decades of research on mitigating both power side-channel attacks and timing attacks.

Fortunately, despite its significance, Hertzbleed has only been demonstrated on one specific cryptographic primitive, SIKE. That attack crucially took advantage of SIKE's inner workings to construct ciphertexts that trigger a sustained low-power state in the CPU depending on a secret bit. With SIKE now deprecated due to unrelated security concerns [2]–[4], Hertzbleed's security implications are unclear.

In this paper, we initiate the study of Hertzbleed after SIKE. Through three case studies, we show that Hertzbleed's effects extend beyond SIKE to other cryptosystems; beyond cryptography to other software that handles secret data; and beyond the CPU cores to data processed elsewhere on a system-on-chip (SoC).

Our first two case studies concern cryptography. In both, we show that Hertzbleed gives a new way to reveal compromising information also targeted in previous side-channel attacks: for ECDSA, the number of leading 0 bits in the nonce (cf. [5]–[11]); for Classic McEliece, whether the number of errors exceeds the correction bound (cf. [12]–[14]). The implementations we study (BearSSL for ECDSA, the optimized reference implementation for Classic McEliece) are hardened against timing side channels, yet we identify novel ways in which they are vulnerable to Hertzbleed.

Our attack against BearSSL's ECDSA implementation is on the edge of practicality, and notable mostly for breaking an extremely carefully written implementation of a standardized and well-studied cryptosystem. By contrast, our second attack, against Classic McEliece, is practical, and we demonstrate full plaintext recovery via 17.5 days of interaction with the server across a LAN.

One limitation of both case studies is that ECDSA signing and Classic McEliece decapsulation are too fast to saturate the CPU (to reach thermal limits) with a request-per-TCP-connection server. For the sake of demonstration, we sidestep this limitation with a server that multiplexes multiple requests in a single TCP connection; we discuss this design decision further below.

In our third case study, we show that Hertzbleed can be used to violate the same-origin policy in the latest version of Google Chrome due to data-dependent power consumption *in the integrated GPU* (iGPU) when applying SVG filters to graphical data. This attack is completely practical, recovering pixel values cross origin at between 1 and 3 pixels per second across half a dozen Intel and AMD machines we tested. Along the way we demonstrate, for the first time, that iGPUs exhibit data-dependent power consumption, and that variable power draw in one SoC component (the iGPU) can cause frequency throttling in another (the CPU cores). Notably, ours is the first demonstrated pixel stealing attack in which the SVG filter rendering code runs in the iGPU and where the filter's running time is the same regardless of leaked pixel color.

Our core contributions are:

- We demonstrate that there exist cryptographic implementations *beyond SIKE* vulnerable to Hertzbleed.
- We demonstrate that Hertzbleed can be induced by computations occurring *outside the CPU cores*.
- We demonstrate how to leverage this cross-component Hertzbleed effect to perform cross-origin pixel stealing in Google Chrome, entirely from JavaScript.

Disclosure. We disclosed our findings to Intel, AMD, Google, BearSSL and the Classic McEliece team. BearSSL acknowledged our findings. The Classic McEliece team did not respond. As of April 2023, the Chrome developers were still deciding whether and how to patch.

2. Background: Hertzbleed

Modern processors dynamically adjust their CPU frequency to reduce power consumption (during low CPU load) or to ensure that thermal parameters remain below safe limits (during high CPU load). This feature is commonly known as dynamic voltage and frequency scaling (DVFS). Hertzbleed attacks leverage the discovery that, during high CPU loads, DVFS-induced frequency adjustments depend on the data being computed on. This is because these adjustments depend on the CPU power consumption, which is data dependent. Moreover, data-dependent frequency changes directly translate to execution time differences (as 1 hertz = 1 cycle/second), which are remotely observable. Specifically, Wang et al. demonstrated that computing on data with high Hamming Weight (HW) or Hamming Distance (HD) causes the CPU to run at a lower frequency than computing on data with low HW or HD [1]. They exploited this observation to perform a remote key extraction on constant-time implementations of SIKE, a post-quantum key encapsulation mechanism. They concluded that "current cryptographic engineering practices for how to write constant-time code are no longer sufficient to guarantee constant time execution of software". Despite its significance, however, this claim has only been shown to hold against SIKE. Moreover, Hertzbleed was only ever applied to leak secrets in the CPU.

3. Beyond SIKE: The Possibility of Triggering Hertzbleed in Other Cryptosystems

We now demonstrate, for the first time, that the implications of Hertzbleed to constant-time cryptography extend beyond SIKE. To this end, we show that the constant-time implementations of two other prominent cryptosystems are also affected by Hertzbleed. The first is the Elliptic Curve Digital Signature Algorithm (ECDSA), a widely-used classical public key cryptosystem. The second is Classic McEliece, an emerging post-quantum cryptosystem that recently advanced to the Fourth Round of NIST's Post-Quantum Cryptography (PQC) competition. Without taking Hertzbleed into account on modern CPUs, cryptographic implementations following current constant-time programming principles can fail to achieve true constant-time execution.

Table 1: CPUs used in Section 3's experimental setup.

CPU Model	Cores	Base Frequency	Max Frequency
AMD Ryzen 7 4800U (Zen 2)	8	1.80 GHz	4.20 GHz
Intel Core i7-8700 (Coffee Lake)	6	3.20 GHz	4.60 GHz
Intel Core i7-12700K (Alder Lake)	12	3.60 GHz	4.80 GHz

3.1. Experimental Setup

We run our experiments on three machines, one with an AMD processor and two with an Intel processor, whose details are shown in Table 1. All machines run Ubuntu 22.04 with Linux 5.15 and the latest microcode patches installed. To monitor CPU frequency and package domain power consumption, we use the same approach as Wang et al. [1]. For the CPU frequency, we use the `MSR_IA32_MPERF` and `MSR_IA32_APERF` registers. For the power consumption, we use the running average power limit (RAPL) interface [15, §15.10]. When monitoring CPU frequency, we use the default system configuration. Our i7-12700K implements Intel's recently introduced data operand independent timing (DOIT) mechanism [16]. On this machine, we run CPU frequency experiments twice, once with DOIT disabled and again with DOIT enabled. When monitoring CPU power consumption (package domain), we disable frequency boost so that the CPU runs at the base frequency. When reporting average CPU frequency, power, or execution time, we first filter out the outliers and then compute the average.

Some of our experiments rely on separate client and server processes. The primitives we study in this section are too fast for a simple TCP-connection-per-request server to saturate our CPUs, unlike SIKE's decapsulation. We considered carefully tuning the network stack a la Shenango [17], but settled instead on a simple connection-multiplexing server. The client establishes multiple TCP connection to the server, opens multiple logical streams within each connection (using the Go Yamux library), and sends multiple requests sequentially within each stream. We do not claim that any deployed server uses this configuration, but we do expect that other server configurations in which the client can saturate the CPU would show similar results.

The target server and the attacker are both connected to the same network, and we measure an average round-trip time of between the two machines.

3.2. Case Study on ECDSA

ECDSA is a popular signature scheme with an unfortunate vulnerability: A leak of the *nonce*—the ephemeral randomness—used in generating a signature reveals the long-term signing key. To compensate for this vulnerability, implementations of the ECDSA signing operation must be carefully written to avoid leaking information about the nonce over a side channel. In this section, we show that the implementation of ECDSA signing in an especially conservative cryptographic library, BearSSL, leaks information about the nonce through Hertzbleed. Ironically, the

programming tricks used to make the signing code constant-time are the source of the power difference that makes the Hertzbleed attack possible.

Background. ECDSA is an elliptic curve signature scheme, and inherits the following global parameters from the curve over which it is defined:

- $E(\mathbb{F}_q)$: An elliptic curve E over a finite field $\mathbb{F}_q$.
- n: A large prime that divides the order of $E(\mathbb{F}_q)$.
- G: An element of $E(\mathbb{F}_q)$ that has order n.

An ECDSA keypair consists of a secret signing key $d \in \mathbb{Z}/n\mathbb{Z}$ and a public verification key $Q = [d]G$.

Given a nonce k and a message to sign whose hash is h (both quantities in $Z/n\mathbb{Z}$), the signing algorithm first computes $U \leftarrow [k]G$. It then sets $r \leftarrow (U)_x \bmod n$, where the notation $(\cdot)_x$ means the x coordinate of the argument. Finally, it computes $s \leftarrow k^{-1}(h + dr) \bmod n$. The ECDSA signature is (r, s).

Extracting the secret key with a known nonce. Crucially, knowledge of the nonce k used to generate a signature (r, s) would allow an attacker to reconstruct the signing key d as $(sk-h)/r \bmod n$. This means that even a temporary failure of a system's random number generator could compromise its long-term secret.[1]

One approach for hardening ECDSA against randomness failures is to *derandomize* it: To generate the nonce by applying a cryptographic pseudorandom function (PRF) to the message. RFC 6979 specifies a concrete instantiation of this hardening measure, with HMAC as the PRF [19].

A consequence of derandomizing ECDSA that is important for our attack is that the signer will use the same nonce every time it signs a given message.

Extracting the secret key with known nonce bits. The leak of even partial information about nonces can undermine ECDSA security. A line of beautiful cryptanalytic results shows how an attacker who obtains a handful of signatures whose nonces all have their most significant few bits equal to 0 can recover the secret key. For 256-bit curves, just 64 signatures with nonces whose top 4 bits are 0, or just 32 signatures with nonces whose top 8 bits are 0, suffice for recovering the signing key [5]. (Other leakage patterns are also dangerous; see Heninger [20].)

A naive implementation of scalar multiplication leaks the length of the scalar multiplier (i.e., the number of leading 0 bits) via a timing side channel. That leak makes such an implementation unsuitable for computing $[k]G$ as part of ECDSA signing. ECDSA implementations use a variety of mitigations to close this potential side channel, for example padding k by adding multiples of n until it is a fixed length; see Weiser et al. [21] for a survey.

Scalar multiplication in BearSSL's ECDSA implementation. Below, we examine the implementation of ECDSA in BearSSL's latest development version.[2] For concreteness, we describe BearSSL's "`m64`" implementation (the default on 64-bit platforms) of arithmetic on the NIST P-256 curve.

On this curve, BearSSL implements scalar multiplication using a fixed 4-bit window left-to-right double-and-add algorithm, shown in Listing 1, which is extracted (and rewrapped) from `ec_p256_m64.c`. The function `point_mul_inner` can be used to compute $[k]P$ for any scalar k and any point P in constant time. Its argument `W` holds the 4-bit window lookup table $[1]P, [2]P, \ldots [15]P$; when P is the public curve generator G, as is the case for ECDSA signing, this table is precomputed.

The loop maintains an accumulator point `Q`. An iteration of the main loop (listing lines 15–64) computes `Q` $\leftarrow [16]$`Q`$+$ $[$`bits`$]P$, where `bits` is a four-bit nibble extracted from the scalar k (line 28).

The code uses a series of clever tricks to implement this in constant time while accounting for edge cases.

First, in a naive implementation, we would start the loop at the most significant nonzero nibble of k and, in that iteration, set $Q \leftarrow [\textit{bits}]P$. This would, of course, not be constant time. Instead, `point_mul_inner` tracks whether Q is uninitialized using the flag `qz`. So long as the flag `qz` is set, `Q` holds the all-0 bitpattern. (The projective point doubling function, `p256_double`, is written in such a way that doubling an all-0 input produces an all-0 output.)

Second, to extract $[$`bits`$]P$ from the array `W` without using a secret value as an array index, the function iterates through every entry of `W`, using bitwise masks to extract only the entry `W[n]` for which `n`$+1 =$ `bits` (lines 36–47). When `bits` $= 0$, the mask condition is never satisfied, and `T` is never updated from the all-0 bitpattern with which it was initialized. The all-0 bitpattern is not the bit representation of any affine point on the curve, so in the case `bits` $= 0$ the `p256_add_mixed` operation that takes `T` as an input (line 50) does not produce meaningful output.

Third, masking logic using the `qz` and `bnz` flags ensures that, at the bottom of the loop, the accumulator `Q` is set to the right value:

1) If `Q` holds a meaningful value (`qz` is unset) and `T` holds a meaningful value (`bnz` is set), then the conditional copy at line 61 will overwrite `Q` with `U`, which holds the sum of `Q` and `T`.
2) If `Q` holds a meaningful value (`qz` is unset) but `T` does not (`bnz` is unset), then the conditional copy will not overwrite `Q`, leaving it at 16 times its value in the previous loop iteration due to the four `p256_double` invocations (lines 24–27).
3) If `Q` does not yet hold a meaningful value (`qz` is set) but `T` does (`bnz` is set), then the mask `m` is set to all-1 (line 55), causing `Q`'s x and y coordinates to be copied from `T` (lines 57–58) and its z coordinate to be set to (the Montgomery representation of) 1 (line 59). In this case the flag `qz` is cleared (line 62).

1. Indeed, the PlayStation 3's signing key was recovered because its ECDSA implementation used predictable nonces [18].

2. Commit `46f7dddce75227f2e40ab94d66ceb9f19ee6b1b0`, 8 June 2022.

```
static void point_mul_inner(p256_jacobian *R,
  const p256_affine *W,
  const unsigned char *k, size_t klen)
{
  p256_jacobian Q;
  uint32_t qz;

  memset(&Q, 0, sizeof Q);
  qz = 1;
  while (klen -- > 0) {
    int i;
    unsigned bk;

    bk = *k ++;
    for (i = 0; i < 2; i ++) {
      uint32_t bits;
      uint32_t bnz;
      p256_affine T;
      p256_jacobian U;
      uint32_t n;
      int j;
      uint64_t m;

      p256_double(&Q);
      p256_double(&Q);
      p256_double(&Q);
      p256_double(&Q);
      bits = (bk >> 4) & 0x0F;
      bnz = NEQ(bits, 0);

      /* Lookup point in window. If the bits
       * are 0, we get something invalid,
       * which is not a problem because we
       * will use it only if the bits are
       * non-zero. */
      memset(&T, 0, sizeof T);
      for (n = 0; n < 15; n ++) {
        m = -(uint64_t)EQ(bits, n + 1);
        T.x[0] |= m & W[n].x[0];
        T.x[1] |= m & W[n].x[1];
        T.x[2] |= m & W[n].x[2];
        T.x[3] |= m & W[n].x[3];
        T.y[0] |= m & W[n].y[0];
        T.y[1] |= m & W[n].y[1];
        T.y[2] |= m & W[n].y[2];
        T.y[3] |= m & W[n].y[3];
      }

      U = Q;
      p256_add_mixed(&U, &T);

      /* If qz is still 1, then Q was
       * all-zeros, and this is conserved
       * through p256_double(). */
      m = -(uint64_t)(bnz & qz);
      for (j = 0; j < 4; j ++) {
        Q.x[j] |= m & T.x[j];
        Q.y[j] |= m & T.y[j];
        Q.z[j] |= m & F256_R[j];
      }
      CCOPY(bnz & ~qz, &Q, &U, sizeof Q);
      qz &= ~bnz;
      bk <<= 4;
    }
  }
  *R = Q;
}
```

Listing 1: BearSSL P-256 scalar multiplication implementation, `m64` variant.

4) Finally, if `Q` does not yet hold a meaningful value (`qz` is set) and `T` also does not (`bnz` is unset), none of the masking operations changes the value of `Q`.

In ECDSA signing, the `point_mul_inner` function is always called with a 32-byte (256-bit) `k`; the above tricks ensure that its running time is the same regardless of how many leading all-0 nibbles `k` has, under a traditional Kocher-style analysis that rules out secret-dependent branching, indexing, and variable-time instructions [22].

BearSSL ECDSA under power analysis. However, the *power usage* of the `point_mul_inner` function varies depending on how many leading all-0 nibbles `k` has.

The variable `Q` is initialized to the all-0 bitpattern, and holds that value until a non-zero nibble is encountered. The point `T` extracted from the window also holds the all-0 bitpattern in every loop iteration where `bits` $= 0$.

As a result, each additional leading all-0 nibble in `k` induces four additional `p256_double` calls whose input is all-0, an extra `p256_add_mixed` call both of whose inputs are all-0, and some additional bitwise operations with all-0 inputs in the `point_mul_inner` loop. Every intermediate value produced in these subroutine calls is also all-0. The result is that leading all-0 nibbles trigger ALU dataflows with low HW/HD: The more leading all-0 nibbles k has, the lower the power consumption.

Because scalar multiplication is by far the most computationally intensive step in ECDSA signing, a significant power difference in scalar multiplication induces a significant power difference for the signing operation as a whole.

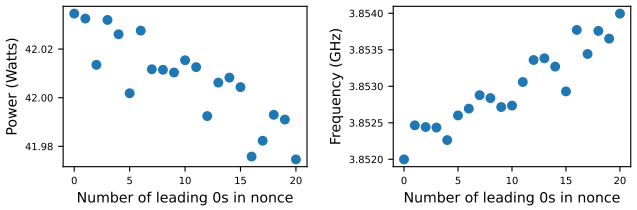
BearSSL ECDSA under Hertzbleed. Crucially, BearSSL's ECDSA signing is derandomized using RFC 6979. As a result, *every* time a given message is signed, the same nonce is chosen. If we repeatedly request signatures from a multithreaded signing server for a message that happens to be associated with a nonce with many top all-0 nibbles, *all* the signing operations will have low power draw. If we repeatedly request signatures for a message associated with a nonce whose top nibble is nonzero, *all* the signing operations will have high power draw.

As we show below, this accumulated power difference is significant enough to induce timing variability under Hertzbleed. As a result, BearSSL's ECDSA signing routine is not constant time in practice despite following the Kocher principles for constant-time programming.

Experimental validation. We use the setup of Section 3.1 to locally monitor CPU frequency and power consumption when BearSSL ECDSA generates signatures with nonces containing 1 to 20 leading 0s. We sample power/frequency every and collect 1,000,000 data points per experiment.

Our Proof-of-Concept (PoC) has a randomly generated secret key d. For each count i of leading 0 bits, we create 3 random messages $\{m_{i,0}, m_{i,1}, m_{i,2}\}$ whose RFC 6979 nonces have exactly i leading 0 bits.

We launch a multithreaded BearSSL signing server. The server spawns pthreads on all CPU cores. Each thread



(a) CPU power consumption vs number of leading 0s in k

(b) CPU frequency vs number of leading 0s in k

Figure 1: Average CPU power consumption and frequency on our i7-8700 CPU while BearSSL ECDSA signs messages that generate nonces k with different numbers of leading 0s.

handles the signing request with $m_{i,j}$, s as input and runs in an infinite loop for the duration of the experiment. Figure 1 shows the results on our i7-8700 CPU, grouped according to the number of leading 0s in k.

First, our results demonstrate that the CPU power consumption during BearSSL ECDSA's signature generation depends on the number of leading 0s in k. In Figure 1a, we confirm that the CPU power consumption decreases as the number of leading 0s in k increases.

Second, our results demonstrate that the above power leakage translates to a frequency leakage under Hertzbleed. In Figure 1b, we confirm the CPU frequency increases as the number of leading 0s in k increases. We observe consistent results across all our tested processors and regardless of DOIT mode status.

Next, we show that the frequency signal is also visible remotely as a timing difference. We set up server and client following Section 3.1 with the i7-8700 as our server. Our server is embedded with a long-term signing secret key. The server reads in the message, performs the signing computation, and sends the signature back to the client.

The client targets $1, \ldots, 20$ leading 0s of k using the messages generated above. For each message $m_{i,j}$, the client spawns 12 TCP concurrent connections. Each TCP connection starts 200 logical streams, and each logical stream sends out 100 signing requests in a loop. The client times how long it takes the server to finish $12 \times 100 \times 200 = 120{,}000$ signing requests and collects 500 samples. The client interleaves all $m_{i,j}$ to even out unusual behavior on the server.

Figure 2 shows the results grouped according to the number of leading 0s. These results demonstrate that non-constant-time signal is remotely observable. The signing time decreases when the number of leading 0 bits in k increases because the server CPU frequency increases.

Discussion. While the timing differences shown above are clear, they are small in absolute terms and compared to measurement noise. As a result, we do not claim that remote key extraction is practical, even with a server that can be induced to perform ECDSA signing on all hardware threads. Our implementation would take over 200 days to collect enough messages with 8 leading 0 bits to recover the key.

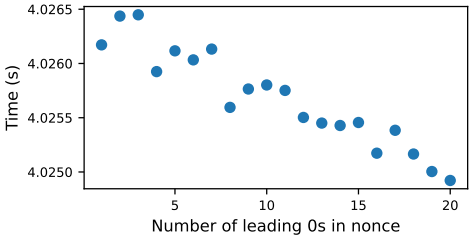


Figure 2: Remote timing vs number of leading 0s in the nonce k using BearSSL ECDSA on our i7-8700 CPU.

3.3. Case Study on Classic McEliece

Classic McEliece is a round-4 submission to NIST's Post-Quantum Cryptography Standardization Project that is related, but not identical, to the (1978) McEliece cryptosystem [23]. It builds a key encapsulation mechanism (KEM) from the Niederreiter cryptosystem, which is a dual version of the original McEliece cryptosystem and is specified to be implemented using binary Goppa codes [24].

The original McEliece system is vulnerable to an adaptive chosen-ciphertext attack (CCA) called Sloppy Alice [12], which relies on a decoding failure oracle to recover the plaintext. Classic McEliece claims to be not vulnerable to this attack due to the implicit rejection in the KEM that checks plaintext re-encryption against the ciphertext [24].

We show the specific implementation choice of the Classic McEliece's NIST submission contains a decoding failure oracle under power analysis, which can be turned into timing analysis under Hertzbleed. As a result, there exists timing leakage in Classic McEliece even though its implementation follows the current constant-time programming principles.

Background. The computational objects on which Classic McEliece is built are error-correcting codes. (By contrast, ECDSA is built on elliptic curves.)

Specifically, Classic McEliece uses *binary Goppa codes*. A binary Goppa code is an (n, k) linear code $\mathcal{C}$ over a finite field $\mathbb{F}_{2^m}$ and is a k-dimensional ($k = n - mt$) subspace of the vector space $\mathbb{F}_{2^m}^n$; here t is a crucial parameter described next. For any pair of codewords $(\mathbf{x}, \mathbf{y})$ in $\mathcal{C}$, the distance between them, $\text{dist}(\mathbf{x}, \mathbf{y})$, is the Hamming distance. The weight of $\mathbf{x} \in \mathbb{F}_{2^m}^n$, $\text{wt}(\mathbf{x})$, is its Hamming weight.[3]

A binary Goppa code $\mathcal{C}$ as above has error-correction capacity t, which means that a receiver can recover a codeword that has been corrupted in up to t positions, but not in $t + 1$ or more. More precisely, given a codeword $\mathbf{c}$ and an arbitrary vector $\mathbf{e}$, if $\text{wt}(\mathbf{e}) \leq t$, there is an efficient decoding algorithm $\mathcal{A}$ that always recovers $\mathbf{c}$ from $\mathbf{c} \oplus \mathbf{e}$, but if $\text{wt}(\mathbf{e}) > t$, it fails to recover $\mathbf{c}$ [25].

A binary Goppa code $\mathcal{C}$ has an $(n-k) \times n$ parity check matrix H such that $H\mathbf{c}^T = 0$ for any $\mathbf{c}$ in $\mathcal{C}$. For an arbitrary vector $\mathbf{c}' \in \mathbb{F}_2^n$, the syndrome of $\mathbf{c}'$ is $H\mathbf{c}'^T \in \mathbb{F}_2^{n-k}$.

A Classic McEliece private key comprises a binary Goppa code $\mathcal{C}$ and a random string s used when the re-

3. Details about binary Goppa codes can be found in Appendix A.

Algorithm 1: Classic McEliece adaptive CCA given a decoding failure oracle.

Input: C, H, Decoding failure Oracle
Output: Plaintext $\mathbf{e}$

$\mathbf{e} = \{\}$
for $i \leftarrow 0$ **to** n **do**
  $C' \leftarrow C \oplus H[i]$
  if *Oracle*(C') $\neq$ *Failure* **then**
    $\mathbf{e} \leftarrow \mathbf{e} \cup \{i\}$
return $\mathbf{e}$

encryption check fails. After reducing the parity-check matrix H of $\mathcal{C}$ to the systematic form $[I_{n-k}|T]$, the public key is T, a matrix of size $(n-k) \times k$. (Here I_{n-k} is the $(n-k) \times (n-k)$ identity matrix.)

Encapsulation picks a random vector $\mathbf{e} \in \mathbb{F}_2^n$ with $\mathrm{wt}(\mathbf{e}) = t$. Then it recovers H as $[I_{n-k}|T]$ from the public key T and generates the ciphertext $C = H\mathbf{e} \in \mathbb{F}_2^{n-k}$; the session key K is the hash of $\mathbf{e}$ as described below.

Decapsulation takes in the secret key and a ciphertext C and outputs the session key K as follows:

1) Extend C to $\mathbf{v} = (C, 0, \ldots, 0) \in \mathbb{F}_2^n$ by appending k zeros.
2) If $\mathbf{v}$ is within the error-correction capacity of the decoding algorithm $\mathcal{A}$, $\mathcal{A}$ recovers $\mathbf{e}'$ as $\mathbf{v} \oplus \mathbf{c}$ where $\mathbf{c}$ is a unique codeword such that $\mathrm{dist}(\mathbf{c}, \mathbf{v}) \leq t$.[4] Otherwise $\mathcal{A}$ recovers $\mathbf{e}'$ as failure $\perp$.
3) Check $\mathrm{wt}(\mathbf{e}') = t$ and that re-encrypting $\mathbf{e}'$ outputs C.
4) If the checks fail, compute $K = \mathrm{Hash}(0, s, C)$. Otherwise, compute $K = \mathrm{Hash}(1, \mathbf{e}', C)$.

For the attacks we describe, it is important to remember that ciphertexts generated by the encapsulation algorithm have error vector $\mathbf{e}$ with weight *exactly* t and the decoding subroutine $\mathcal{A}$ invoked in step 2 of the decapsulation has a sharp transition in behavior: It will recover a codeword (and error vector) with at most t errors but will fail to recover any codeword with $t+1$ or more errors. For more details on Classic McEliece, we refer to its NIST submission [24].

An Adaptive CCA on Classic McEliece with a decoding failure oracle. Classic McEliece is vulnerable to an adaptive CCA if given a decoding failure oracle [13]. With such an oracle, an adversary who knows a ciphertext can recover the plaintext. The underlying idea originates from the Sloppy Alice attack on the original McEliece cryptosystem [12].

Suppose the adversary knows a ciphertext $C = H\mathbf{e}$ and wishes to recover the corresponding weight-t error vector $\mathbf{e}$ and thus the plaintext. If the adversary can manipulate C to C' by adding or removing an error, they can induce the sharp transition of behavior in the decoding subroutine described above. If C' has $t-1$ errors, the decoding subroutine will output $\mathbf{e}'$ with $\mathrm{wt}(\mathbf{e}') = t-1$. By contrast, if C' has $t+1$ errors the decoding subroutine will fail (and output $\perp$) because $t+1$ exceeds the error-correction capacity of $\mathcal{A}$.

4. Since $\mathbf{v} = \mathbf{c} \oplus \mathbf{e}'$, $\mathbf{e}' = \mathbf{v} \oplus \mathbf{c}$.

```
// If decode succeeds, roots of locator are
// positions of 1s in plaintext e'; otherwise,
// locator is a random polynomial.
bm(locator, s_priv);
fft(eval, locator);

// re-encryption and weight check
// If decoding succeeds, error256 contains
// wt(e') nonzero entries; otherwise,
// error256 contains very few nonzero entries.
allone = vec256_set1_16b(0xFFFF);
for (i = 0; i < 16; i++)
{
  error256[i] = vec256_or_reduce(eval[i]);
  error256[i] = vec256_xor(error256[i],
↪  allone);
}
// Expand error256 to scaled for bitslicing,
// which amplifies the difference in the
// number of non-zero entries.
scaling_inv(scaled, inv, error256);
// Input scaled to transposed FFT to compute
// H*error256, which further amplifies the
// difference in HW/HD.
fft_tr(s_priv_cmp, scaled);

// Check the validity of re-encryption, but it
// is too late, because the Hertzbleed signal
// is already generated.
check_synd = synd_cmp(s_priv, s_priv_cmp);
```

Listing 2: Classic McEliece re-encryption code, annotated.

A decoding failure oracle is some way (for example, through a side channel) for the adversary to learn whether or not $\mathcal{A}$ outputs $\perp$ given input C of the adversary's choice. If the adversary has access to a decoding failure oracle, they can extract $\mathbf{e}$ bit by bit as follows. The key observation is that, if C is a ciphertext corresponding to an error vector $\mathbf{e}$, then $C' = C \oplus H[i]$ is a ciphertext corresponding to the error vector $\mathbf{e}'$ that agrees with $\mathbf{e}$ in all positions except i; here $H[i]$ is the ith column of the parity-check matrix H, which the adversary can reconstruct given the public key. If $\mathbf{e}[i] = 1$, then $\mathbf{e}'[i] = 0$, so $C' = C \oplus H[i]$ has $t-1$ errors. By contrast, if $\mathbf{e}[i] = 0$, then $\mathbf{e}'[i] = 1$, so $C' = C \oplus H[i]$ has $t+1$ errors. Given C', a decoding failure oracle will distinguish these two cases, and reveal whether $\mathbf{e}[i]$ is 0 or 1. We provide details about this CCA in Appendix B.

A naive plaintext recovery attack based on the above observation is shown in Algorithm 1 and can be optimized by almost a factor of 10 as shown in prior work [13].

Classic McEliece decapsulation under power analysis. The Classic McEliece NIST submission implementations are meant to keep the adversary from obtaining a decoding failure oracle by a timing side channel. However, we show that three of its four implementations—`avx`, `sse`, and `vec`—nevertheless allow the adversary to infer decoding failure through Hertzbleed. A specific implementation choice shared by these implementations but not the unvectorized implementation `ref`—the use of bitslicing—causes step 3 of decapsulation (the re-encryption check) to run with high HW/HD data if the decoding succeeds, but low HW/HD data

if the decoding fails. When the same ciphertext is repeatedly decapsulated, the different power draw between these data patterns triggers different processor frequency profiles and therefore different decapsulation timings.

Listing 2 shows the re-encryption checks in the `avx` implementation of the `mceliece348864` parameter set (which has $n = 3488$, $t = 64$, and $m = 12$). The code shown is excerpted from the `decrypt` function, with comments added for annotation. We confirmed that we can also trigger similar timing leakage in the other two optimized (`sse` and `vec`) implementations for this parameter set.

Classic McEliece uses the constant-time Berlekamp-Massey (BM) algorithm for decoding [26]. Given a ciphertext $C = H\mathbf{e}'$, the BM algorithm returns an error-locator polynomial `locator` (listing line 4). The algorithm proceeds to check the validity of `locator` via re-encryption.

When decoding succeeds, the roots of `locator` identify the positions of 1s in $\mathbf{e}'$. When decoding fails, `locator` is a random polynomial, which most likely has a few roots over $\mathbb{F}_{2^m}$. The `locator` polynomial is evaluated on all field elements using a fast Fourier Transform (FFT) (line 5). The entries of `eval` are scanned to identify those that are 0 (i.e., are roots of `locator`) and the results are saved into the boolean array `error256` (lines 11–16). If $\mathrm{wt}(\mathbf{e}') \leq t$, the decoding succeeds, and `error256` has $\mathrm{wt}(\mathbf{e}')$ non-zero entries. If $\mathrm{wt}(\mathbf{e}') > t$, the decoding fails, and `error256` has a few non-zero entries.[5]

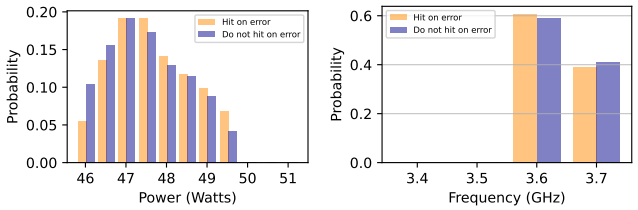
Then, at line 20, `error256` is converted to `scaled`, a 2d array bitsliced representation of elements in $\mathbb{F}_{2^m}$ [28]. Bitslicing is a data orthogonalization technique that stores a x-bit `data` into x variables by spreading each bit of `data` into the i-th bit of each x variables. Bitsliced representations are suitable for constant-time implementations because they decompose complicated operations into a series of bitwise operations. However, bitslicing creates a significant power leakage here because it amplifies the HW/HD difference in `error256` into a bigger one. Compared to a successful decoding, a decoding failure creates on average $t-2$ fewer non-zero entries in `error256`. However, in `scaled` the difference is amplified to around $m(t-2)/2$ because bitslicing spreads one data element into multiple machine words.

Even worse, at line 24, `scaled` is passed to a transposed FFT, `fft_tr`, whose butterfly subroutine recursively combines the results of lower-level butterflies into higher-level butterflies. Any HW/HD difference in the `fft_tr` input will be further amplified as the butterfly expands [28].

As a result, the re-encryption subroutine exhibits a meaningful power leakage that reveals whether the decoding failed. With such a decoding failure oracle we can extract the plaintext following Algorithm 1.

Finally, at line 29, the algorithm checks the validity of re-encryption, but this does not mitigate the power leakage because it comes in too late.

5. Experimentally we find `error256` in this case contains at most 2–3 non-zero entries. A random polynomial over a finite field has one root in expectation [27].



(a) CPU power vs $H[i]$ (b) CPU frequency vs $H[i]$

Figure 3: Power consumption and frequency on our i7-8700 CPU while running Classic McEliece's decapsulation on $C' = C \oplus H[i]$ with $H[i]$ hitting or not on hitting an error.

Experimental validation. We use the setup of Section 3.1 to locally monitor CPU frequency and power consumption while running Classic McEliece's decapsulation with different input ciphertexts. We sample power/frequency every 1 ms and collect $2,000,000$ data points per experiment.

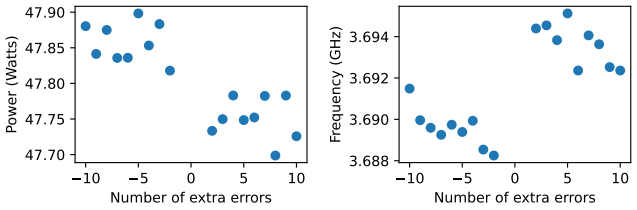
We launch a multithreaded Classic McEliece decapsulation server with a static secret key. The server spawns pthreads on all CPU cores. Each thread handles a decapsulation request with a ciphertext and the secret key as input and runs in an infinite loop for the duration of the experiment.

Starting from a valid ciphertext $C = H\mathbf{e}$ (known to the adversary), we construct 2 groups of malformed ciphertexts:

1) G_1 reflects our naive CCA in Algorithm 1. G_1 consists of ten different ciphertexts $C' = C \oplus H[i]$. Five of them have $H[i]$ that hits on an entry of $\mathbf{e}$. They do not trigger the decoding failure because the number of errors that C' contains is 1 below the error-correction capacity t. The other five have $H[i]$ that does not hit on an entry of $\mathbf{e}$. They trigger the decoding failure because the number of errors that C' contains is 1 above t.
2) G_2 reflects the optimized CCA as shown by Lahr et al. that queries the decoding failure oracle on malformed ciphertexts C' containing as many as x errors below or above the capacity t [13]. (For the parameter set we target, $x = 10$.) G_2 contains eighteen different C' such that nine of them contain $t-10$ to $t-2$ errors (inducing a decoding failure) and the other nine contain $t+2$ to $t+10$ errors (not inducing a decoding failure).

Figure 3 shows the results for G_1, grouped according to whether or not $H[i]$ hits on an error. When $H[i]$ does not hit on an error, the decoding failure is triggered, and the decapsulation produces a data flow with lower HW/HD (due to the bitslicing in re-encryption) compared to when $H[i]$ hits on an error. As a result, the power consumption decreases and the CPU frequency increases.

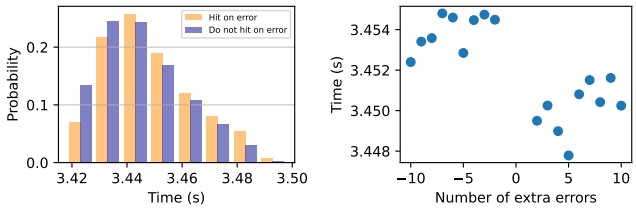
Figure 4 shows the results for G_2. Again, when C' introduces 2 to 10 extra errors, the decapsulation consumes less power and runs at a higher CPU frequency compared to when C' removes 2 to 10 existing errors. The gap between the two cases in both the power consumption and CPU frequency suggests that the power/frequency leakage due to triggering decoding failure can be distinguished regardless of the extra number of errors being introduced or removed



(a) CPU power vs Number of extra errors

(b) CPU frequency vs Number of extra errors

Figure 4: Average power consumption and frequency on our i7-8700 CPU while running Classic McEliece's decapsulation on C' such that C' contains -2 to -10 or 2 to 10 extra errors compared to C.



(a) G_1: Remote timing leakage vs $H[i]$

(b) G_2: Remote timing leakage vs Number of extra errors in C'

Figure 5: Remote timing for the naive CCA (simulated by G_1) and the optimized CCA by Lahr et al. [13] (simulated by G_2), using Classic McEliece on our i7-8700 CPU.

by C'. We observe consistent results across all our tested processors and regardless of DOIT mode status.

Classic McEliece decapsulation under Hertzbleed. We demonstrate that the above frequency leakage is even visible remotely. We set up a server and a client following section 3.1 with the server running on our i7-8700 CPU. The server is embedded with a long-term secret key. The server reads in the malformed ciphertext C' and performs the decapsulation, after which it sends an acknowledgment back to the client.

The client uses the same G_1 and G_2 groups of C' as above. For each C', the client spawns 12 TCP concurrent connections. Each connection starts 500 logical streams, and each logical stream sends out 100 decapsulation requests in a loop. The client times how long it takes the server to finish $12 \times 500 \times 100 = 600{,}000$ decapsulation requests and collects 500 data points. The client interleaves all C' to even out unusual behavior on the server machine.

Figure 5 shows the results grouped by C'. In Figure 5a, we show that there is a remote timing leakage for our naive CCA in Algorithm 1. When the number of errors in C' is one above the capacity t ($H[i]$ does not hit on an error), the decoding failure is triggered and the decapsulation time decreases because the server CPU frequency increases. In Figure 5b, we show that there is a remote timing leakage for the optimized CCA of Lahr et al. [13]. In terms of decapsulation time, the gap between C' removing x number of errors, and C' introducing x number of errors suggests that the timing leakage due to decoding failure is robust.

Plaintext recovery. We validate the apparent leakage by performing a full plaintext recovery on the optimized `avx` implementation of Classic McEliece with the parameter set `mceliece348864` and following a similar approach to Lahr et al. We profile the server and determine a timing threshold for decoding failures using the original ciphertext that triggers 64 errors and a malformed ciphertext that does not trigger any error. We collect 300 data points per query and re-run a query when the result is inconclusive. After 17.5 days we had recovered 62 error positions in the plaintext and got the remaining two by brute-force search.

4. Beyond Cryptography and CPU Core Data: Leaking Web Browser Secrets from the iGPU

We introduce the first Hertzbleed attack where data-dependent *CPU* frequency differences are induced by computations occurring *outside the CPU core* Specifically, we demonstrate, for the first time, that CPU frequency adjustments are affected by the integrated GPU (iGPU) power consumption. Additionally, the iGPU power consumption depends on the data values being computed on in the iGPU. As a result, CPU frequency adjustments depend on the data values being processed in the iGPU. We use this observation to construct a pixel stealing attack on the latest version of Google Chrome that allows attackers to recover content in a cross-origin iframe due to iGPU power differences created by SVG filters. Ours is the first cross-origin pixel stealing attack after Chrome offloaded all SVG filter rendering to the GPU, and where the SVG filter stack itself does not exhibit any timing variance based on input values.

In independent and concurrent work, Taneja et al. show that (integrated and discrete) GPU power consumption can trigger *GPU* frequency changes [29] in integrated and discrete GPUs from multiple vendors. Like us, Taneja et al. construct a pixel stealing attack against Chrome. Our attack and Taneja et al.'s attack both induce pixel-color–dependent power consumption in the GPU, but differ in how the attacker measures the power consumption difference. In our attack, CPU frequency varies, so the attacker measures the running time of a synthetic JavaScript workload running on the CPU (see Section 4.5); in Taneja et al.'s attack, the GPU frequency varies and so, therefore, does GPU workload running time, so the attacker can directly query filter running time (using `requestAnimationFrame`).

4.1. Background: Pixel Stealing Attacks

A Web browser manages secrets on behalf of its user with each of many sites the user visits. Some (but not all) of these secrets influence the visible content of site pages rendered by the browser. For example, the main Gmail page shows the contents of a logged-in user's inbox.

The web sites a user visits are mutually distrusting. The browser must maintain the isolation of each site even while

Table 2: CPUs and iGPUs used in Section 4.3's experimental setup.

CPU Model	iGPU Model	CPU Cores	CPU Base Frequency	CPU Max Frequency	iGPU Base Frequency	iGPU Max Frequency	Package TDP
Intel i7-8700 (Coffee Lake)	UHD 630	6	3.20 GHz	4.60 GHz	350 MHz	1.20 GHz	65 W
Intel i7-9700 (Coffee Lake Refresh)	UHD 630	8	3.00 GHz	4.70 GHz	350 MHz	1.20 GHz	65 W
Intel i7-10510U (Comet Lake)	UHD 620	4	1.80 GHz	4.90 GHz	300 MHz	1.15 GHz	15 W

executing JavaScript programs provided by one site that invoke APIs implemented by browser components that have access to other sites' secrets. The same-origin policy defines the ways in which sites can interact and the ways in which they must be isolated [30].

One example of an allowed interaction is framing. A page on one site can incorporate a page on another site using an iframe, even though the same-origin policy prevents it from reading the other page's contents. The browser renders the enclosing page and enclosed page and composites them. Using cascading stylesheets (CSS), the enclosing page can ask the browser to apply complex graphical transformations to the enclosed page. Some available transformations are defined as part of the CSS specification; others are incorporated by reference from the structured vector graphics (SVG) specification.

A cross-origin leak is a vulnerability that allows one site's JavaScript program to obtain another site's secret in violation of the same-origin policy. *Pixel stealing* is a cross-origin leak in which the secret revealed is the visual content of a target site page. Pixel stealing, and cross-origin leaks generally, may arise either from a browser implementation bug or from a side channel [31]–[36].

In 2013, Stone [33] (and, concurrently, Kotcher et al. [34]) showed how to use side channels in browser rendering code for pixel stealing. Stone observed that the Firefox implementation of the `feMorphology` SVG filter included a pixel-value–dependent branch, and showed that this branch could be used as a timing side channel for pixel stealing. In Stone's attack, the enclosing page uses a stack of CSS filters that ask the browser's rendering engine to extract a single pixel from the enclosed iframe, duplicate it onto a larger surface, composite the duplicated-pixel surface with another attacker-supplied image, and then apply the vulnerable `feMorphology` filter to the composited surface. The attacker-supplied image is chosen so that, when the target pixel is white, Firefox's `feMorphology` filter implementation takes one branch, and, when the target pixel is black, it takes the other (faster) branch. The duplicated-pixel surface serves to amplify the timing difference; it is sized so that processing the filter stack takes more than when the input is a white pixel, but less than when the input is a black pixel. The attack page uses the browser's `requestAnimationFrame` API to register a function to be called when page rendering completes; by observing when this callback is invoked, the attacker site can deduce whether the target pixel was white or black.

Andrysco et al. [35] and Kohlbrenner and Shacham [36] showed that some SVG filter implementations in Firefox, Chrome, and Safari used floating-point instructions whose runtime depends on their operand values. They used Stone's attack framework to mount pixel-stealing attacks on these browsers, again relying on `requestAnimationFrame` to distinguish slow target pixels from fast.

Browser vendors have attempted to patch these vulnerabilities by rewriting their SVG filter implementations to run in constant time, using techniques originally developed for implementing cryptographic routines [22]. These techniques are discussed in the W3C Filter Effect specification [31] and the numerous bug reports on pixel stealing via non-constant-time SVG filters or CSS operations [32], [37], [38].

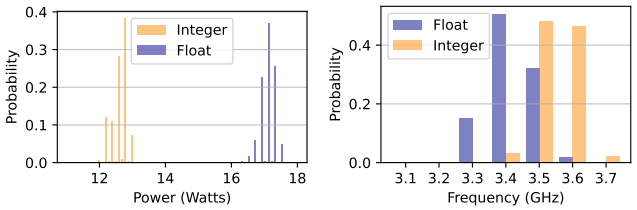
4.2. Experimental Setup

We run our experiments on three machines with different Intel processors, the details of which are presented in Table 2. All our machines run Ubuntu 20.04 or 22.04 with kernel 5.4 or 5.15, the latest microcode patches and the latest Google Chrome version (107 64-bit) installed. Our display resolution is 1920×1080. To monitor CPU frequency, we use the methodology from Section 3.1. To monitor iGPU power consumption, we do `perf_event_open` system calls to the PMU events `power/energy-gpu`. To monitor iGPU frequency, we use `perf_event_open` system calls to the PMU events from the Intel i915 driver, as done in Linux [39]. We sample CPU/iGPU frequency and iGPU power consumption every 5 ms. To lock the CPU frequency (when we explicitly mention doing so), we set `max_perf_pct` and `min_perf_pct` in `/sys/devices/system/cpu/intel_pstate` to the same value, following the intel_pstate driver [40].

4.3. iGPU-CPU Frequency Leakage Channel

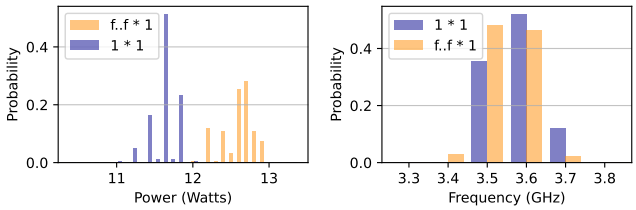
We now show that, on modern Intel processors, the distribution of a CPU's frequency values leaks information about the workloads being executed on the iGPU as well as the data being computed on in the iGPU.

GPU workloads from OpenCL. A GPU has clusters of cores containing processing elements. OpenCL allows the application to specify a kernel, which is an algorithm expression for how a single work-item will be executed on a single processing element. Each work-item, identified by a global ID, is a thread with respect to its control flow and its memory model [41]. Multiple work-items can be grouped together to a work-group to be scheduled to execute on a core. OpenCL exposes the maximum number of work-items that a work-group is capable of executing on a core via `CL_DEVICE_MAX_WORK_GROUP_SIZE`.



(a) iGPU power vs iGPU workload (b) CPU frequency vs iGPU workload

Figure 6: Dependency between iGPU workload, iGPU power, and CPU frequency on our i7-8700. When the CPU workload is fixed, the iGPU workload (`Integer`) that consumes less power causes a higher CPU frequency compared to the iGPU workload (`Float`) that consumes more power.



(a) iGPU power vs iGPU data (b) CPU frequency vs iGPU data

Figure 7: Dependency between data being processed in the iGPU, iGPU power, and CPU frequency on our i7-8700. Computing on data with a lower HW consumes less power on the iGPU, which translates to a higher CPU frequency than computing on data with a higher HW.

We construct two different 1-dimensional OpenCL workloads, one called `Integer` with a kernel of integer instructions and the other one called `Float` with a kernel of floating-point instructions. To utilize the GPU's parallel processing architecture, we partition the global work of both workloads into work-groups of size equal to `CL_DEVICE_MAX_WORK_GROUP_SIZE`.

CPU frequency is iGPU workload-dependent. To start, we set out to understand if and how running different iGPU workloads affects the CPU frequency values when the CPU is under high load. To this end, we run the above two OpenCL workloads while concurrently running `matrixprod` from the `stress-ng` benchmark suite [42] on all CPU cores. This setup ensures that the CPU is under a high load (and therefore leaks via Hertzbleed) while the iGPU workload is running. We collect $600,000$ iGPU power consumption, iGPU frequency and CPU frequency data points during the execution of each experiment.

Figure 6a shows the iGPU power consumption when running both workloads on our i7-8700 processor. Not surprisingly, the iGPU power consumption depends on the iGPU workload. The `Integer` iGPU workload consumes less power compared to the `Float` one. What is interesting from our perspective is that this difference in iGPU power consumption influences CPU frequency. This effect is visible in Figure 6b, which shows the CPU frequency when the CPU workload was fixed to `matrixprod` and only the iGPU workload varied. When the iGPU consumes less power, the CPU runs at a higher frequency, and vice versa.

CPU frequency is iGPU data-dependent. Similarly, we find that CPU frequency depends on the specific *values* being computed on in the iGPU with a fixed kernel.

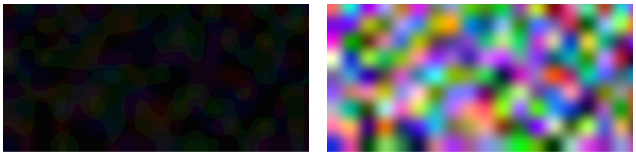
We use the same `Integer` iGPU workload as above and only change the data values being computed on, keeping the rest of the experiment setup used for Figure 6 the same. In the first case, the OpenCL kernel repeatedly computes `1 × 1` whereas in the second case the OpenCL kernel repeatedly computes `0xffffffffffffffff × 1`.

Figure 7a shows the iGPU power consumption when running `Integer` with both inputs on our i7-8700 processor. We immediately see that the data being processed in the iGPU impacts the power consumption of the iGPU. The `1 × 1` case, with a lower HW, consumes less power compared to the `0xffffffffffffffff × 1` case, with a higher HW. Figure 7b shows the CPU frequency (with a CPU workload of `matrixprod`) during the same experiment varying iGPU input values to the `Integer` kernel. As expected, when the iGPU power consumption is lower the CPU runs at a higher frequency, and vice versa. We believe that this result is purely due to the data-dependent power consumption of the iGPU. First, we constructed our workloads in a way that the interaction between the CPU and the iGPU is identical regardless of the input. Second, we also monitored the iGPU frequency during each experiment, and observed that it did not vary based on the input.

4.4. SVG Filters and CPU Frequency

We saw that the data being computed on in the iGPU can lead to differences in CPU frequency. We now demonstrate how this behavior can be leveraged by a web attacker by constructing an SVG-filter pixel-stealing attack [33] that measures CPU frequency differences arising from iGPU data differences. Broadly, the attacker selects a cross-origin pixel from an iframe, fills another iframe with the pixel's color value and executes an SVG filter stack that can cause CPU frequency differences depending on the pixel's color.

SVG filter stack framework. An SVG filter stack is a series of graphical computations that can be applied to elements of a web page (including cross-origin elements, see 4.1) to create visual effects. Chrome runs these computations on the GPU by default (i.e., with hardware acceleration enabled). There are more than a dozen available SVG filters in Google Chrome including Gaussian blurs, color transforms, image compositions and generalized convolutions. An important stepping stone toward end-to-end pixel stealing is to find an SVG filter stack which, when rendering on an iframe filled with an attacker-selected cross-origin pixel, creates iGPU power consumption differences that depend on the pixel's color. As we showed in the previous section, these iGPU power differences can trigger CPU frequency differences. Using our method described in Section 4.5, an attacker can then observe these differences to infer the pixel's color.



(a) black: `0.01 × random` (b) white: `0.99 × random`

Figure 8: Compositions of `iframe` and `random` with `feComposite` when the target pixel is black and white.

To analyze our proposed SVG filter stack, we build a framework that consists of multiple web pages and native CPU frequency and iGPU power sampling. On the main web page, we embed a cross-origin web page with only one black or white pixel into an iframe.[6] To isolate and expand this target pixel within the iframe, we perform the following operations (handled by the GPU):

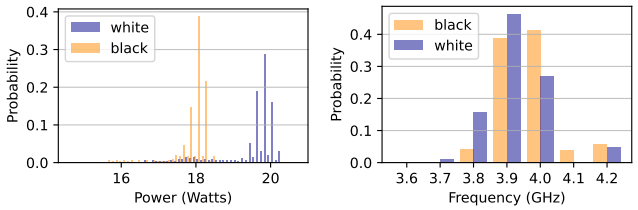
1) We use the CSS property `clip-path` to create a 1×1 clipping region for the target pixel.
2) We use the `scale` function in the CSS property `transform` to expand the 1×1 region into a 2000×2000 `iframe`, which is either all black or all white.

Next, we construct our proposed SVG filter stack `FilterStack` to be applied onto the above `iframe`. Specifically, `FilterStack` generates a meaningful iGPU power difference depending on the target pixel being black or white. `FilterStack` works as follows:

1) We generate an image with random colors of size 20×20. Then we scale it to 2000×2000, the same size as the target `iframe`. We use this scaled image `random` as our source of randomness.
2) We use `feComposite` to compose `random` and the target `iframe`. We choose the operation `arithmetic` which computes each resulting pixel using the following formula: $k_1 i_1 i_2 + k_2 i_1 + k_3 i_2 + k_4$, where i_1 indicates the pixel channel of `iframe` and i_2 indicates the pixel channel of `random`. We set $k_1 = 0.98$, $k_2 = 0$, $k_3 = 0.01$, and $k_4 = 0$. The composed image, shown in Figure 8, is `0.01 × random` (almost all-black with low HW/HD) if the target pixel is black and `0.99 × random` (random noise with high HW/HD) if the target pixel is white.
3) Now with the above composed image as our source, we finish our `FilterStack` by appending GPU heavy filter `feGaussianBlur` in a chain with standard deviation set to 0.999. This works as an amplifier of the HW/HD difference in our composed images, and its iGPU power consumption should be very distinct depending on the target pixel being black or white.

Experimental result. We apply our `FilterStack` repeatedly on the `iframe` targeting either a white or a black pixel in an infinite loop of `requestAnimationFrame`. Simultaneously, we stress the CPU by launching one JavaScript worker thread per CPU core that infinitely does integer multiplication in the browser. This setup ensures that the CPU is under a high load while the iGPU `FilterStack` renders. We collect $200,000$ iGPU power consumption and CPU frequency data points during each experiment.

6. An arbitrary color can be binarized to black and white using `feColorMatrix` and `feComponentTransfer`.



(a) iGPU power vs target pixel being black or white (b) CPU frequency vs target pixel being black or white

Figure 9: Dependency between target pixel color, iGPU power, and CPU frequency on our i7-8700 (Chrome 107). Our `FilterStack` creates a substantial iGPU power difference, which translates to a CPU frequency difference, when the isolated target pixel is black versus white.

```
let prev = Date.now();
while (true) {
    curr = Date.now();
    if (prev != curr) {
        frequency_array.push(inc);
        inc = 0;
        prev = curr;
    }
    inc += 1;
}
```

Listing 1: Browser CPU frequency sampler.

Figure 9a shows the iGPU power consumption on our i7-8700 (Chrome 107) when the target pixel is black versus white. Since our `FilterStack` creates a source image with high (target pixel: white) versus low (target pixel: black) HW/HD, and keeps stressing the iGPU to amplify such a difference, we observe substantial iGPU power difference depending on the pixel color. This result shows that our `FilterStack` is effective at creating iGPU power consumption differences that depend on the cross-origin pixel color. Figure 9b demonstrates that, as with previous experiments, CPU frequency indirectly reflects the color of the pixel targeted by the `FilterStack`.

4.5. Measuring CPU Frequency from JavaScript

In a typical web attacker threat model, the attacker does not have access to native interfaces to monitor the CPU frequency. Thus, the last step for constructing an end-to-end pixel stealing attack is to find a method to indirectly measure CPU frequency from the browser. We solve this challenge by constructing a frequency sampler that reliably measures CPU frequency.

Browser CPU frequency sampler. To construct our frequency sampler, we re-use the framework of Section 4.4.

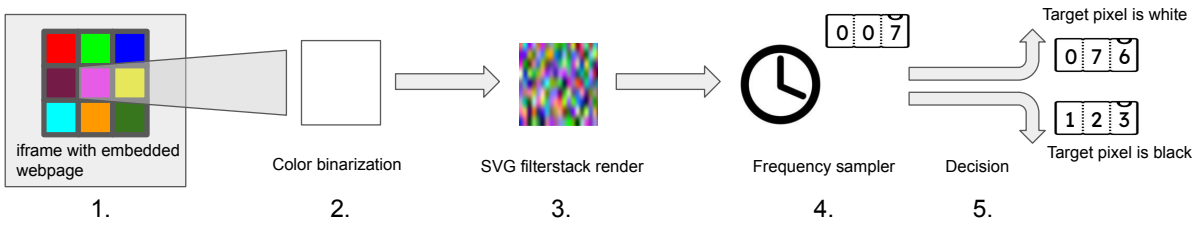


Figure 10: Pixel stealing Proof-of-Concept steps.

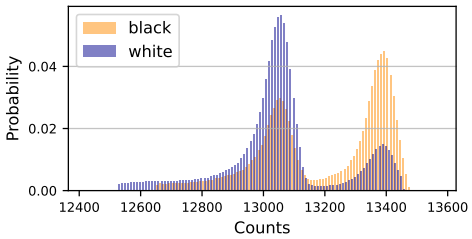


Figure 11: Frequency sampler readings on our i7-8700, when stressing the CPU cores and alternately rendering `FilterStack` on black and white pixels every 1 second.

Figure 12: Pixel stealing PoC result: Original vs Reconstruction on our i7-8700 processor.

However, instead of using native code to monitor the CPU frequency directly, we leverage the fact that CPU frequency affects the number of JavaScript operations that can be completed within a fixed time interval.

In our frequency sampler, shown in Listing 1, we increment a counter in a loop until the clock reading updates (similarly to prior work [43]). Once the clock reading updates, we save the current counter to an array, reset the counter to zero and repeat. This way, we get an array of counters after a run, with each counter representing the relative CPU frequency between two clock edges. We run our frequency sampler in a separate JavaScript worker thread.

Surprisingly, instantiating Listing 1 with the nominally higher-resolution `performance.now` clock in place of `Date.now` yields a *less* effective frequency sampler in Chrome. We explain why in Appendix C.

Browser frequency sampler and `FilterStack`. Now we confirm that the frequency sampler works as expected when `FilterStack` renders.

We use the framework from Section 4.4, stressing our CPU cores from the browser, rendering `FilterStack` and alternating the targeted pixel between black and white every 1 second for 2000 times. We simultaneously run our `Date.now` frequency sampler in another JavaScript worker.

In Figure 11, we observe that our sampler captures the CPU frequency difference induced by the pixel-dependent iGPU power consumption of `FilterStack`.

To confirm our findings, we checked that applying `FilterStack` on black or white pixels caused no meaningful difference in direct rendering time (approximately 61.2ms for both) and no difference in iGPU frequency.

4.6. Pixel stealing Proof-of-Concept

In the above sections, we established all the components needed to induce a frequency leakage channel for pixel color differences and measure it from JavaScript. We now present an end-to-end pixel stealing attack.[7]

First, we profile the CPU frequency while rendering `FilterStack` on two known cross-origin all white and all black web pages to establish thresholds. Then, our Proof-of-Concept attack on Chrome follows the steps of Figure 10:

1) We embed the target page into a region of size 48×48. We scroll a 1×1 `div` over the region pixel by pixel.
2) For each target pixel, `div` binarizes its color to black or white with `feColorMatrix` and `feComponentTransfer`, and then expands it to a 2000×2000 `iframe` following Section 4.4.
3) Then, while stressing all CPU cores from the browser, we render the `FilterStack` for x seconds on top of the second `iframe` to create substantial iGPU power difference depending on the target pixel color.
4) In another thread, our frequency sampler monitors the CPU frequency between `Date.now` updates. (Approximately every 1 ms).
5) After x seconds, we collect our CPU frequency data and compare to our derived thresholds.

Proof-of-Concept result. Using this PoC we successfully reconstruct visual information belonging to a cross-origin domain. We demonstrate synthetic (checkerboard) and text examples in Figure 12. We use this checkerboard pattern because it has long stabilization periods and abrupt transitions which demonstrates rapid detection of transitions. On our i7-8700, we set the PoC to render the `FilterStack` on each pixel for 0.8 seconds. As shown in Figure 12,

7. We will open source our PoC after Chrome deploys a patch.

Table 3: Details of the devices where we tested the pixel stealing PoC of Section 4.6 and results of running the PoC.

CPU Model	iGPU Model	Device Model	Operating System	Monitor Resolution	PoC Throughput	PoC Error Rate
Intel i7-8700 (Coffee Lake)	UHD 630	Dell XPS 8930	Ubuntu 22.04 (kernel 5.15)	1920 × 1080	0.86 seconds/pixel	0.65%
Intel i7-8700 (Coffee Lake)	UHD 630	Dell XPS 8930	Windows 10 Enterprise	1920 × 1080	2.59 seconds/pixel	4.68%
AMD Ryzen 7 4800U (Zen 2)	Radeon Vega 8	SimplyNUC Ruby R8	Ubuntu 22.04 (kernel 5.15)	1920 × 1080	2.88 seconds/pixel	1.95%
Intel i7-10510U (Comet Lake)	UHD 620	CTL Chromebox CBx2-7	ChromeOS 107	3840 × 2160	2.50 seconds/pixel	9.46%
Intel i7-9750H (Coffee Lake)	UHD 630	Dell XPS 15 7590	Windows 11 Home	1920 × 1080	3.07 seconds/pixel	3.56%
Intel i7-10850H (Comet Lake)	UHD 630	Dell Latitude 5511	Windows 10 Pro	1920 × 1080	2.96 seconds/pixel	2.43%
Intel i7-10510U (Comet Lake)	UHD 620	Lenovo ThinkPad X1	Ubuntu 22.04 (kernel 5.15)	1920 × 1080	1.46 seconds/pixel	2.47%

our PoC reconstructs the checkerboard with a throughput of 0.86 seconds per pixel and an error rate of 0.06%, and reconstructs the "*S&P*" from an iframe of the IEEE S&P 2023 website with a throughput of 1.13 seconds per pixel.

We also test our PoC targeting the checkerboard on a variety of machines all with Chrome version 107 (64-bit) and detail our results in Table 3. This does require tweaking the parameters on different machines (e.g., `div` size, number of layers of `FilterStack`). In general, our PoC works across operating systems and hardware vendors with an error rate under 10% and throughput of 1-3 seconds/pixel (full time extraction 0.5 hour to 1.5 hours).

We find that the difference in performance depends on the amount of DVFS oscillation. Intuitively, thermally constrained devices are more affected by Hertzbleed. On some of our 15W-TDP laptops DVFS oscillates between a large range of P-states, increasing measurement variance.

Full color extraction. Like previous pixel-stealing attacks, ours uses the `feColorMatrix` to transform an input pixel's RGB values to the scalar $xR + yG + zB + c$ for binarization. The attack as evaluated above sets $x = 0.21$, $y = 0.72$, $z = 0.07$, and $c = 0.0$ to extract (roughly) luminance. Alternative coefficient settings allow us to isolate a single-color channel (for example to isolate red by setting $x = 1.0$ and $y = z = 0.0$), and to extract that color channel's exact value by repeated measurements while scaling that color's coefficient and adjusting the constant c.

We have confirmed that this technique allows us to perform a binary search to recover a chosen color channel of a given cross-origin pixel.

5. Mitigations

Our ECDSA attack takes advantage of the derandomizing nature of BearSSL's ECDSA implementation: given a message, the nonce is deterministically generated. To protect against such an attack, one can either use a randomized ECDSA implementation or employ a constant-time implementation strategy that doesn't suffer from power leakages.

Our Classic McEliece attack leverages a commonly known power leakage in the code-based cryptography community, which is the decoding failure oracle. We are not aware of an efficient mitigation of this attack for this implementation style. Replacing the bitslicing technique in reencryption with an alternative is likely required.

To prevent our pixel stealing attack, browsers are the best positioned to make changes. There are several options: disable cross-origin iframe access to sensitive information such as cookies, restrict or remove the application of graphical transforms on cross-origin information, or re-implement the SVG filters to be resilient against power leakage.

In the long term, research is needed to discover generic Kocher-like principles that software can apply to mitigate Hertzbleed, in addition to avoiding secret-dependent branching, indexing, or variable-time instructions.

6. Conclusion

Like speculative execution, DVFS will not be going away any time soon. We are then presented with a problem: power usage is dependent on data, frequency is increasingly dependent on power, and frequency is relatively easy for adversaries to measure. To make things worse, the well-known compromise of trading timing attack resistance for power leakage collapses in the presence of frequency attacks. Time *is* power under the right circumstances.

In this paper, we demonstrated that Hertzbleed's effects are wide ranging, extending beyond SIKE, beyond cryptography and beyond CPU-only secrets. Our cryptographic attacks on ECDSA and Classic McEliece typify how a small power channel can transform into timing channel on well-written code that runs in a constant number of CPU cycles. Our SVG-filter pixel-stealing attack on web browsers demonstrates not only a similar leakage against classically constant-time code, but also how to *indirectly* measure frequency outside of the target code path. Critically, this demonstrates that a defensive approach cannot rely on blinding the timing of the target operations alone, and must consider all sources of frequency leakage.

A common aspect of our attacks worth considering is the large multiplicative factor they rely on; to induce a meaningful Hertzbleed signal we must repeat target operations a significant number of times, often in parallel. While this is currently a challenge for the attacker, as DVFS grows more fine-grained and reactive these limitations will fade. Hertzbleed attacks will get better with each new generation of hardware and power-saving techniques.

Our results suggest that, similarly to Spectre attacks [44], Hertzbleed may continue to haunt us for some time to come.

Acknowledgment

We thank our anonymous reviewers for their valuable feedback. This work was funded by NSF grants 1942888 and 1954521 and gifts from Google, Mozilla, and Qualcomm.

References

[1] Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power side-channel attacks into timing attacks on x86," in *USENIX Security*, 2022.

[2] W. Castryck and T. Decru, "An efficient key recovery attack on SIDH," in *EUROCRYPT*, 2023.

[3] D. Robert, "Breaking SIDH in polynomial time," in *EUROCRYPT*, 2023.

[4] L. Maino, C. Martindale, L. Panny, G. Pope, and B. Wesolowski, "A direct key recovery on SIDH," in *EUROCRYPT*, 2023.

[5] M. R. Albrecht and N. Heninger, "On bounded distance decoding with predicate: Breaking the "lattice barrier" for the Hidden Number Problem," in *EUROCRYPT*, 2021.

[6] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom, "LadderLeak: Breaking ECDSA with less than one bit of nonce leakage," in *CCS*, 2020.

[7] J. Jancar, V. Sedlacek, P. Svenda, and M. Sys, "Minerva: The curse of ECDSA nonces: Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces," in *CHES*, 2020.

[8] D. F. Aranha, P.-A. Fouque, B. Gérard, J.-G. Kammerer, M. Tibouchi, and J.-C. Zapalowicz, "GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias," in *ASIACRYPT*, 2014.

[9] J. Breitner and N. Heninger, "Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies," in *FC*, 2019.

[10] E. D. Mulder, M. Hutter, M. E. Marson, and P. Pearson, "Using Bleichenbacher's solution to the Hidden Number Problem to attack nonce leaks in 384-bit ECDSA," in *CHES*, 2013.

[11] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels," in *CCS*, 2016.

[12] E. R. Verheul, J. M. Doumen, and H. C. van Tilborg, "Sloppy Alice attacks! Adaptive chosen ciphertext attacks on the McEliece public-key cryptosystem," in *Information, coding and mathematics*. Springer, Heidelberg, Germany, 2002, pp. 99–119.

[13] N. Lahr, R. Niederhagen, R. Petri, and S. Samardjiska, "Side channel information set decoding using iterative chunking," in *ASIACRYPT*, 2020.

[14] A. Shoufan, F. Strenzke, H. G. Molter, and M. Stöttinger, "A timing attack against Patterson algorithm in the McEliece PKC," in *ICISC*, 2009.

[15] *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3B: System Programming Guide, Part 2*, Mar. 2023.

[16] "Data operand independent timing instruction set architecture (ISA) guidance," Online: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html, Feb. 2023, accessed on Apr 6, 2023.

[17] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *NSDI*, 2019.

[18] fail0verflow, "Console hacking 2010: Ps3 epic fail," Presented at 27C3. Online: https://media.ccc.de/v/27c3-4087-en-console_hacking_2010, Dec. 2010.

[19] T. Pornin, "Deterministic usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)," RFC 6979, Aug. 2013.

[20] N. Heninger, "RSA, DH and DSA in the wild," in *Computational Cryptography: Algorithmic Aspects of Cryptology*, ser. London Mathematical Society Lecture Note Series. Cambridge University Press, Feb. 2022, vol. 469, ch. 6, pp. 140–81.

[21] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, "Big numbers—big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations," in *USENIX Security*, 2020.

[22] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO*, 1996.

[23] R. J. McEliece, "A public key cryptosystem based on algebraic coding theory," *Deep Space Network Progress Report*, vol. 4244, pp. 114–116, 1978.

[24] M. Albrecht, D. Bernstein, T. Chou, C. Cid, J. Gilcher, T. Lange, V. Maram, I. von Maurich, R. Misoczki, R. Niederhagen, K. Paterson, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, C. J. Tjhai, M. Tomlinson, and W. Wang, "Classic McEliece: conservative code-based cryptography," National Institute of Standards and Technology, Tech. Rep., 2022.

[25] D. J. Bernstein, "Understanding binary-Goppa decoding," Cryptology ePrint Archive, Report 2022/473, 2022.

[26] E. R. Berlekamp, *Algebraic coding theory (revised edition)*. World Scientific, 2015.

[27] V. K. Leont'ev, "Roots of random polynomials over a finite field," *Mathematical Notes*, vol. 80, no. 1, pp. 300–04, 2006.

[28] T. Chou, "McBits revisited," in *CHES*, 2017.

[29] H. Taneja, J. Kim, J. J. Xu, S. van Schaik, D. Genkin, and Y. Yarom, "Hot pixels: Frequency, power, and temperature attacks on GPUs and ARM SoCs," 2023, preprint, arXiv:2305.12784 [cs.CR].

[30] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch, Nov. 2011.

[31] "W3c filter effects module level 1," https://www.w3.org/TR/filter-effects-1/#priv-sec, accessed on Dec 2, 2022.

[32] "Canvas composite operations and CSS blend modes leak cross-origin data via timing attacks," https://bugs.chromium.org/p/chromium/issues/detail?id=699028, accessed on Dec 2, 2022.

[33] P. Stone, "Pixel perfect timing attacks with HTML5," Context Information Security, White Paper, 2013.

[34] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson, "Cross-origin pixel stealing: Timing attacks using CSS filters," in *CCS*, 2013.

[35] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P*, 2015.

[36] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *USENIX Security*, 2017.

[37] "SVG filter timing attack," https://bugzilla.mozilla.org/show_bug.cgi?id=711043, accessed on Dec 2, 2022.

[38] "Pixelstealing and history-stealing through floating-point timing side channel," https://bugzilla.mozilla.org/show_bug.cgi?id=1131288, accessed on Dec 2, 2022.

[39] Linux, "`i915_pmu.c`," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/i915/i915_pmu.c, accessed on Nov 1, 2022.

[40] "Intel P-State driver," https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt, accessed on Nov 21, 2022.

[41] Khronos OpenCL Working Group, *The OpenCL Specification, Version v3.0.13*, Feb. 2023, online: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[42] C. I. King, "stress-ng," https://github.com/ColinIanKing/stress-ng, 2022, accessed on Jun 7, 2022.

[43] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *USENIX Security*, 2016.

[44] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P*, 2019.

Appendix A.
Binary Goppa Codes

A binary Goppa code $\Gamma(\mathbf{L}, g(x))$, parameterized by an irreducible binary Goppa polynomial of degree t over $\mathbb{F}_{2^m}$ and a finite subset $\mathbf{L}$ of $\mathbb{F}_{2^m}^n$, is a linear code often used in McEliece-like cryptosystems.

An irreducible binary Goppa polynomial is simply a degree t irreducible polynomial over $\mathbb{F}_{2^m}$ (m and t as positive integers):

$$g(x) = \sum_{i=0}^{t} g_i x^i \in \mathbb{F}_{2^m}[x] \tag{1}$$

We define $\mathbf{L}$ as a finite subset of $\mathbb{F}_{2^m}^n$, so that

$$\mathbf{L} = \{\alpha_1, ...\alpha_n\} \subseteq \mathbb{F}_{2^m}^n \tag{2}$$

and $g(\alpha_i) \neq 0$.

Now, we can define a binary Goppa code, $\Gamma(\mathbf{L}, g(x))$, parameterized over $\mathbf{L}$ and the $g(x)$: $\Gamma(\mathbf{L}, g(x))$ consists of all code vectors $\mathbf{c} \in \mathbb{F}_2^n$ such that:

$$\sum_{i=1}^{n} \frac{c_i}{(x - \alpha_i)} \equiv 0 \mod g(x) \tag{3}$$

The syndrome of an arbitrary vector $\mathbf{w}$ is defined as:

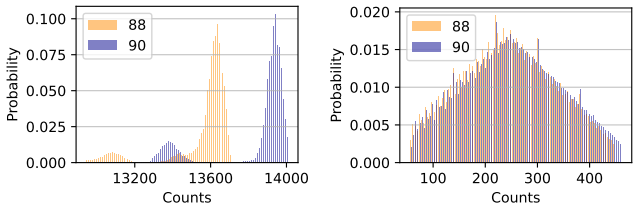
$$S_{\mathbf{w}}(x) = -\sum_{i=1}^{n} \frac{w_i}{g(\alpha_i)} \frac{g(x) - g(\alpha_i)}{x - \alpha_i} \mod g(x) \tag{4}$$

$\Gamma(\mathbf{L}, g(x))$ is a (n, k, d)-code. It can be shown that for $\Gamma(\mathbf{L}, g(x))$, the dimension $k \geq n - mt$ and the minimum distance $d \geq 2t + 1$.

Appendix B.
Details of Classic McEliece CCA

Recall that the ciphertext C is the syndrome of the error $\mathbf{e}$. For a $\mathbf{v}$ in the same coset of $\mathcal{C}$ with $\mathbf{e}$, $C = H\mathbf{e} = H(\mathbf{e}+\mathbf{c}) = H\mathbf{v} \in \mathbb{F}_2^{n-k}$ ($\mathbf{c}$ as an arbitrary codeword). Since H is of the systematic form $[I_{n-k}|T]$, $\mathbf{v}$ can be recovered from C by appending k zeroes to C: $\mathbf{v} = (C, 0, ...0) \in \mathbb{F}_2^n$ because $H\mathbf{v} = C = H(\mathbf{e} + \mathbf{c})$.

Assume the adversary attempts to probe entry i of $\mathbf{e}$. They can create a vector $\mathbf{e}'$ such that it is all 0 except at position i. They compute the new syndrome C' as $H(\mathbf{v} \oplus \mathbf{e}')$, which is essentially $C \oplus H[i]$ ($H[i]$ as the i-th column of H). They send C' to the decoding algorithm $\mathcal{A}$. Observe that $C' = H(\mathbf{v} \oplus \mathbf{e}') = H(\mathbf{e} \oplus \mathbf{e}')$ contains $t-1$ errors if $\mathbf{e}[i] = 1$, and $t+1$ errors otherwise. Due to the error-correction capacity of $\mathcal{A}$, if C' contained $t+1$ errors, it would trigger decoding failure and $\mathcal{A}$ would generate $\perp$, which implies that $\mathbf{e}[i] = 0$. On the other hand, if C' contained $t-1$ errors $\mathcal{A}$ would generate $\mathbf{e} \oplus \mathbf{e}'$, which implies $\mathbf{e}[i] = 1$. Therefore, if the adversary has access to such a decoding failure oracle, they can extract $\mathbf{e}$ bit by bit.



(a) `Date.now` (b) `performance.now`

Figure 13: Counter values of our frequency samplers when locking our i7-8700 CPU frequency to 88% versus 90%.

Appendix C.
CPU Frequency Sampling from JavaScript

The best frequency sampler for our attack should give us distinct counters when the CPU runs at different frequencies. In order to accurately detect the CPU frequency during a short interval, the sampler needs to have high resolution and low overhead for each reading. In terms of resolution, `Date.now` has a resolution of 1 ms and `performance.now` has a resolution of 0.1 ms on Chrome 107 (64-bit). Compared to `Date.now`, we can collect more data points from `performance.now` during a fixed amount of time. However, this advantage is outweighed by the fact that between two clock edges we can call `Date.now` about 5 times more than `performance.now`.

We experimentally confirm the best frequency sampler between the two by locking the CPU frequency to 88% or 90% of the maximum frequency to emulate the difference generated by `FilterStack` and then collecting 200 seconds of measurements from each sampler.

As Figure 13 shows, the frequency sampler from `Date.now` gives a significant counter difference when the CPU is locked to 88% versus 90% (Figure 13a), whereas the frequency sampler from `performance.now` only gives minor counter difference (Figure 13b).

To understand the root cause of this difference, we traced the optimized assembly code of the two timers produced by the JavaScript JIT in Chrome, which is how our timers are run most of the time after the initial JIT warmup of a few thousand iterations. Calling a `Date.now` turns out to be compiled into a single call into a *V8 builtin* which then directly calls into `Runtime_DateCurrentTime` in *V8* internally. In comparison, when a `performance.now` gets called, it needs to call `LoadGlobalICTrampoline` and `CallFunctionTemplate`, both doing complicated type checks by calling into `CheckObjectType` many times, before the `NowOperationCallback` *Blink binding* eventually gets called which then reads the timer. We observe that eventually `Runtime_DateCurrentTime` calls `gettimeofday` and `NowOperationCallBack` calls `clock_gettime` on the Linux platform. Since there is only about a 5% difference between these two timers on our i7-8700, the performance difference in the browser comes from the complexity in the generated code path.

We note that Chrome makes the output of both `Date.now` and `performance.now` fuzzy as Spectre mitigation. The MurmurHash invocation used in timer fuzzing entails significant performance overhead for JavaScript that calls a timer in a tight loop, as ours does. The Chrome developers are now working to reduce the performance overhead of timer fuzzing,[8] which may improve the effectiveness of our frequency sampler.

8. See https://crbug.com/1414615.