License Locking via Time-Based Device Fingerprinting

ANDREI COMAN*, JACOB POLATTY*, and YUNZHOU LI*, Columbia University, USA

In this paper, we explore the potential of using hardware fingerprinting - the use of measurable hardware characteristics to uniquely identify a device - to design a mechanism of license locking. We explore several fingerprinting techniques and implement a modification of the CryptoFP algorithm [Sanchez-Rola et al. 2018] to achieve 100% discrimination power. Our fingerprints are stable over periods of at least one month, and the final lincense-locked executables are resilient against several types of attacks, including tampering with baseline fingerprints, subverting the timing mechanism and system-call replay attacks. We also explore alternative fingerprinting methods, such as the result of a contended write in a race condition between threads.

CCS Concepts: • **Security and privacy → Software reverse engineering**; **Software security engineering**; Key management; **Hardware reverse engineering**.

Additional Key Words and Phrases: Fingerprinting, Hardware Fingerprinting, License Locking

ACM Reference Format:
Andrei Coman, Jacob Polatty, and Yunzhou Li. 2024. License Locking via Time-Based Device Fingerprinting. *ACM Trans. Priv. Sec.* X, X, Article X (May 2024), 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Hardware fingerprinting [Sanchez-Rola et al. 2018] [Salo 2007] [Venugopalan et al. 2023] is a method of uniquely identifying a device via measurable characteristics of the hardware (originating from e.g. CPU execution, DRAM Rowhammer attacks, clocks). Such a technique can be used for recommendation systems and advertisment distribution on the web without the use of cookies [Fin 2024], authentication and attacker identification on the web among others.

Here, we explore the use of hardware fingerprinting in License-Locking - binding the license to an executable to a specific hardware device. This is of great interest as it could avoid many of the common pitfalls of standard software licensing, such as multiple use of a single license, granting the product distributor tighter control over the application use. The remainder of our paper is structured as follows: Section 2 presents a brief overview of existing hardware fingerprinting techniques, Section 3 states our threat model, Section 4 describes our adaptation of the CryptoFP algorothm, Section 5 lists several attacks and mitigations, Section 6 presents our results, and Section 7 discusses alternative fingerprinting methods.

*All authors contributed equally to this research.

Authors' Contact Information: Andrei Coman, coman.andrei@columbia.edu; Jacob Polatty, jacob.polatty@columbia.edu; Yunzhou Li, yl5407@columbia.edu, Columbia University, New York, NY, USA.

ACM 2471-2574/2024/5-ARTX
https://doi.org/XXXXXXX.XXXXXXX

2 BACKGROUND

We are aware of several hardware fingerprinting techniques with potential to be converted into a fast, reliable license-locking system. [Salo 2007] designed a hardware fingerprint by repeatedly measuring the number of CPU clock cycles, as reported by the Timestamp Counter (TSC), that elapsed within a 1-second tick of the Realtime Counter (RTC). A similar fingerprint was based on the comparison between the TSC and the Sound Card Digital Signal Processor (DSP) clock. Although these two methods achieved 98.7% and 93.3% respective discrimination powers, the fingerprint collection was on the order of 1 hour, which might prove prohibitive for the purpose of license locking.

[Venugopalan et al. 2023] based their fingerprinting method on the Rowhammer attack [Kim et al. 2014], by measuring the probability of specific DRAM rows to induce bit-flips in the neighbouring rows, and used the Jensen-Shannon divergence as a similarity measure between fingerprints. This method reports 99.91% fingerprinting accuracy with an associated fingerprinting time of less than 10 seconds. However, the nature of the attack (Rowhammer) is very invasive, architecture dependent, and might lead to crashes on the user side.

[Sanchez-Rola et al. 2018] built on the work of [Salo 2007] and achieved 100% discrimination power by measuring the difference between collocated hardware clocks. The method, called CryptoFP, was presented in both a host-based setting and a browser-based setting. Because of the robustness of the host-based CryptoFP implementation, we have decided to use CryptoFP as a baseline for our License Locking mechanism. The main algorithm is further explored in Section 4.

3 THREAT MODEL

In our threat model, we take the role of the agent performing license-locking on an executable. We assume the agent possesses a secure server which can perform cryptographic computation, and require the user to communicate with this server over the network at fingerprinting time. Our goal is to differentiate between machines with different or identical hardware, while maintaining fingerprint stability on a single machine for extended periods of time.

Additionally, we assume the agent can execute the fingerprinting code natively on the user's machine, and is given access to a specific set of hardware-based clocksources (e.g. TSC, HPET or other crystal oscillators). We evaluate the performance of our fingerprinting code using either the TSC or the HPET clocksource based on the criteria of [Sanchez-Rola et al. 2018].

Lastly, we assume the end-user might attempt to evade the fingerprinting mechanism by either tampering with the baseline fingerprint, subverting system calls entirely, or by mounting a system-call replay attack (based on the knowledge of the time-based nature of our fingerprint), and present mitigations against these attacks. We briefly note that further security can be obtained by code obfuscation.

4 METHODOLOGY

Our base approach closely follows that of [Sanchez-Rola et al. 2018]. For the sake of completeness, we include a brief summary of their method and our adaptation thereof below.

In this setting, a fingerprint $\mathcal{F}$ is constructed as the aggregation of several time-based measurements. The measurements concern the execution time of a *fingerprinting function* f on several inputs $j_1, j_2, ..., j_n$ over m repetitions. More formally, if we let Δ^i denote the time-measurement operator in the i-th repetition,

$$\mathcal{F} = \begin{bmatrix} \Delta^1 f(j_1) & \Delta^2 f(j_1) & ... & \Delta^m f(j_1) \\ \Delta^1 f(j_2) & \Delta^2 f(j_2) & ... & \Delta^m f(j_2) \\ \vdots & \vdots & & \vdots \\ \Delta^1 f(j_n) & \Delta^2 f(j_n) & ... & \Delta^m f(j_n) \end{bmatrix}$$

The number of inputs n is chosen to generate sufficient entropy to differentiate between all relevant machines, and the inputs $j_1, j_2, ..., j_n$ are chosen such that the execution time of f over any of the inputs is small enough to be interrupted by context switches with low probability. Similarly, the number of repetitions m is large enough to guarantee the concentration of measurements around a mode value. Following the guidelines of [Sanchez-Rola et al. 2018], we choose $n = 1000$, $m = 50$, and fingerprinting function $f =$ `std::hash`. The inputs to f are strings of respective lengths 1..1000.

We also attempt to design the time-measurement operators Δ^i as described by [Sanchez-Rola et al. 2018] - that is, based on the difference between two collocated clocks. This should have the desirable consequence of cancelling out any effects that CPU load, temperature and other external, possibly adversarial factors might have on one single clock. The closest alternatives available to us to the Windows `datetime` API referenced in the original paper are the Timestamp Counter (TSC) and the POSIX compliant `clock_gettime` function, which takes as input a `clockid_t` in {`CLOCK_REALTIME`, `CLOCK_MONOTONIC`, `CLOCK_MONOTONIC_RAW`, ...}. Following the observation that quotients of execution times yield better invariance to external factors (and unit conversions), our most faithful reproduction of the original algorithm is:

```
fingerprint F;
for (i = 1; i <= m; i++) {
    for (j = 1; j <= n; j++) {
        clock_gettime(CLOCK_REALTIME, &startTime);
        startTSC ← rdtsc();
        fp_func(j);
        clock_gettime(CLOCK_REALTIME, &endTime);
        endTSC ← rdtsc();
        delta ← sensitivity * (endTime.tv_nsec -
             startTime.tv_nsec) / (endTSC - startTSC);
        F[j - 1][i - 1] ← delta;
    }
}
```

Note here the interspersing of the calls to `clock_gettime` and `rdtsc`, which guarantees that both clocks measure equivalent code-snippets. The performance of this first implementation is explored in Section 6. In short, it fails to differentiate between machines with identical hardware. We attribute this result to the TSC-based nature of the `clock_gettime` system call (Appendix A).

More broadly, we were unable to identify two nanosecond-precision clocksources available from userspace on UNIX-based systems. We also note that, lacking such clocksources, we run the risk of hitting the same runtime-barrier as the method of [Salo 2007].

To avoid this, we implement a second fingerprinting method based on the HPET (High Precision Event Timer) Linux clocksource. Due to the clocksource initialization and priority-setting (Appendix A), this clocksource is typically unavailable from userspace on x86 architectures. Therefore, we require superuser privileges in order to modify the clocksource as follows[1]:

```
sudo bash -c 'echo tsc >
    /sys/devices/system/clocksource/
    clocksource0/current_clocksource'
```

Once the HPET clocksource has been enabled as the Kernel's default clocksource, a second, simpler fingerprinting procedure can then be implemented, based on one single clocksource:

```
fingerprint F;
for (i = 1; i <= m; i++) {
    for (j = 1; j <= n; j++) {
        clock_gettime(CLOCK_REALTIME, &startTime);
        fp_func(j);
        clock_gettime(CLOCK_REALTIME, &endTime);
        delta ← endTime.tv_nsec - startTime.tv_nsec;
        F[j - 1][i - 1] ← delta;
    }
}
```

The results are, once again, summarized in Section 6, but, in short, they are comparable with the ones reported by the host-based implementation of [Sanchez-Rola et al. 2018] (i.e. full discrimination power, even among identical machines).

Lastly, to verify if a baseline fingerprint $\mathcal{F}_B$ matches a re-measured fingerprint $\mathcal{F}$, we perform the following check:

$$\left|\left\{ i \in [n] \middle| \text{mode} \begin{bmatrix} \Delta^1 f(j_1) \\ \Delta^1 f(j_2) \\ \vdots \\ \Delta^1 f(j_n) \end{bmatrix} \in \begin{bmatrix} \Delta^1_B f(j_1) \\ \Delta^1_B f(j_2) \\ \vdots \\ \Delta^1_B f(j_n) \end{bmatrix} \right\}\right| + \left|\left\{ i \in [n] \middle| \text{mode} \begin{bmatrix} \Delta^1_B f(j_1) \\ \Delta^1_B f(j_2) \\ \vdots \\ \Delta^1_B f(j_n) \end{bmatrix} \in \begin{bmatrix} \Delta^1 f(j_1) \\ \Delta^1 f(j_2) \\ \vdots \\ \Delta^1 f(j_n) \end{bmatrix} \right\}\right| \geq 0.5 \cdot 2n$$

That is, for every column of repeated measurements, we check if its mode belongs to its counterpart in the other fingerprint, and declare a match if the fraction of positive set memberships is above a fixed threshold (0.5). As will be discussed in the next section, this matching procedure is complex and does not lend itself to easy encryption.

[1] AWS EC2 Documentation

5 FINGERPRINT SECURITY

In this section, we explore several immediate attack vectors which can subvert the fingerprinting mechanism, and describe the respective mitigations we have implemented. In particular, we discuss the use of inner-product functional encryption of [Abdalla et al. 2015] to protect against fingerprint tampering, and timing validation to protect against system-call replay attacks.

5.1 Functional Encryption

The first attack against the fingerprinting method involves tampering with the baseline fingerprint to induce a false positive match. More specifically, recall that, upon licensing the program, a fingerprint is measured and stored on disk. Then, upon re-execution, the fingerprint is re-measured and compared against the baseline. If the attacker can reverse engineer the fingerprint and adequately modify the baseline, licensing becomes ineffective.

To prevent against this attack vector, we perform fingerprint encryption. At a high level, the baseline fingerprint $\mathcal{F}_B$ is encrypted into a cipher C_B and stored on disk. Then, upon re-execution, the re-measured fingerprint $\mathcal{F}$ is compared against C_B, either directly, or indirectly by encrypting $\mathcal{F}$ into C and performing the comparison in the encrypted space. For any meaningful protection, this should be done without decrypting C_B into main memory. The difficulty arises from the complexity of the fingerprint matching algorithm (Section 4), which must perform a set-membership check for every m measurements of the same input and, therefore, is non-trivial to lift into the encrypted space.

5.1.1 Reduction to Inner-Product. The first step towards encryption is, therefore, reducing the complexity of the matching procedure. In this sense, consider two multisets of m measurements each, $\{x_1, x_2, ..., x_m\}$ and $\{y_1, y_2, ..., y_m\}$, corresponding to the baseline and re-measured fingerprint for a single input respectively, and let their modes be x^* and y^*. By drawing inspiration from Bloom filters, consider a common hash function $h : \mathbb{Z} \to [k]$ used to map each of the two multisets into a k-wide hash table as follows:

```
1  HashMeasurements(x_{0:m}, h, k):
2      H_x ← int[k]
3      for(int j = 0; j < m; j++)
4          H_x[h(x_j)] ← 1;
5      H_x[h(mode(x_{0:m})] ← MODE_WEIGHT;
```

That is, H_x behaves as a single-hash Bloom filter, with the exception of the mode, which receives disproportionate weight. Now, assume k is large enough to achieve perfect hashing (i.e. no collisions between distinct elements, among all x's and y's) and consider the (element-wise) dot-product of H_x and H_y, constructed from $x_{0:m}$ and $y_{0:m}$ with the common hash function h. The non-zero terms of the dot-product can be distinguished as either

(1) products of MODE_WEIGHT and MODE_WEIGHT; by the perfect hashing assumption, this implies $x^* = y^*$ and corresponds to two positive set membership tests.
(2) products of MODE_WEIGHT and 1; this implies one mode belongs to the opposite set of measurements and corresponds to a positive set membership test.
(3) products of 1 and 1; these are common non-mode values in the two sets of measurements and correspond to "noise".

Thus, the dot-product of H_x and H_y is strongly correlated with the number of positive set memberships of measurement modes and, consequently, with the matching score. This can be trivially extended to the dot-product of $\mathcal{H}_B$ and $\mathcal{H}$ (the concatenations of respective hash tables across all n inputs for the base and the re-measured fingerprint; each will then be $n \cdot k$ elements long). The MODE_WEIGHT value can then be set to ensure that the "noise", single set-memberships and double set-memberships contribute to the dot-product at different orders of magnitude. To distinguish between double set memberships and single set memberships, we require

$$\text{MODE\_WEIGHT} \cdot \text{MODE\_WEIGHT} > 2n \cdot \text{MODE\_WEIGHT}$$

(i.e. a single product of the first type must be larger than all possible products of the second type) and, to distinguish between single set memberships, we similarly require

$$\text{MODE\_WEIGHT} > 2n \cdot m$$

The second requirement is strictly stronger than the first one. If it is satisfied, then the matching score can be recovered from the dot-product as follows:

```
1  DotProductToMatch(dp):
2      M̃ ← 2 * (dp / (MODE_WEIGHT * MODE_WEIGHT)) +
3              1 * (dp % (MODE_WEIGHT * MODE_WEIGHT) /
                    MODE_WEIGHT) +
4              0 * (dp % MODE_WEIGHT);
5      return M̃;
```

We note that the MODE_WEIGHT determines the bit-length of the hashed-fingerprint entries and, therefore, features in the time complexity of fingerprint encryption. Hence, it should be set on the order of $2n \cdot m$.

At this point, the perfect-hashing assumption can be relaxed. Perfect hashing can be easily achieved by setting $k = O(m^2)$ [Andoni 2021a] and resampling the hash-function h from a universal hash-family $O(1)$ times. Resampling can be avoided by shifting the randomness from the hash-function to the inputs and setting the constant hidden by big-Oh notation to be large enough. However, the quadratic overhead becomes prohibitive due to its contribution to the encryption runtime. The same guarantees can be obtained in using $O(n)$ space [Andoni 2021b], but the resampling issue is more stringent, memory-alignment issues start to feature, and the optimal setting of the asymptotic constant may still be prohibitive.

Hence, we relax the perfect-hashing requirement by considering $k = 10m$. It is trivially true that positive set-memberships in the original fingerprint are preserved by the hashing procedure. However, false set-memberships may occur. Let $\|y\|_0 = \|y_{0:m}\|_0$ be number of distinct measurements in an m-wide row of the fingerprint matrix. Assume x^*, the mode of $x_{0:m}$ does not belong to $y_{0:m}$ and that h is sampled from an m-wise independent hash family [Wegman and Carter 1981].

Then, the probability that x^* does collides with some element in $y_{0:m}$ under hash function h is

$$1 - \left(1 - \frac{1}{k}\right)^{\|y\|_0}$$

since $\|y\|_0 \leq m$ and h is drawn from an m-wise independent family. As this probability is depends on unknown data, we "poison" every multiset of measurements to increase its size to m. This increases the variance of our final estimator, but guarantees a fixed collision probability of

$$p = 1 - \left(1 - \frac{1}{k}\right)^{m}$$

Then, a false positive set-membership will occur with probability p. If M is the true matching score and $\tilde{M}$ is the matching score of the hashed fingerprint, then

$$\tilde{M} = M + \text{Binomial}\,(2n - M, p)$$

Simple properties of the binomial distribution reveal that

$$\mathbb{E}\left[\tilde{M}\right] = M + (2n - M) \cdot p$$
$$\text{Var}\left[\tilde{M}\right] = (2n - M) \cdot p(1 - p)$$

A final estimator can be obtained from $\tilde{M}$ by correcting its bias

$$\mathbb{E}\left[\frac{\tilde{M} - 2n \cdot p}{1 - p}\right] = M$$
$$\text{Var}\left[\frac{\tilde{M} - 2n \cdot p}{1 - p}\right] = \frac{(2n - M) \cdot p}{1 - p} \leq 2n\frac{p}{1 - p} = \sigma^2$$

This estimator is unbiased and, with high confidence, differs from the true match score by an additive 3σ, with σ as above. With $n = 1000$ and $m = 50$, $\sigma \approx 14.5$, which is acceptable for our application.

5.1.2 Function-Hiding Inner-Product Encryption. The reduction above allows us to consider two fingerprints $\mathcal{F}_B$ and $\mathcal{F}$ whose similarity score is uniquely-determined by (and can be recovered from) the inner-product of the two fingerprints. This operation, as opposed to mode set-membership, is more amenable to encryption. In this sense, we use the result of [Abdalla et al. 2015], as implemented in the CiFEr Library [CiF 2021], which guarantees that decrypting C_B (the encrypted $\mathcal{F}_B$ with a key for $\mathcal{F}$ will reveal only $\langle \mathcal{F}_B, \mathcal{F} \rangle$ and nothing else.

The encryption scheme generates a (secret-key, public-key) pair and encrypts $\mathcal{F}_B$ into C_B using the secret key. Then, a "functional-encryption" key $\mathcal{K}$ is generated from $\mathcal{F}$ using the public key. Decrypting C_B using $\mathcal{K}$ then reveals $\langle \mathcal{F}_B, \mathcal{F} \rangle$ and nothing else.

This can be integrated into the fingerprinting scheme as follows: a user measures their fingerprint locally, then sends it to an encryption server. The encryption server generates the (secret-key, public-key) pair, encrypts the baseline fingerprint $\mathcal{F}_B$, deletes the secret-key forever and returns C_B and the public-key. Then, even if the attacker is able to recover the original fingerprint by reverse engineering the hashing procedure and repeatedly querying dot-products, they will be unable to generate a modified fingerprint, as they do not know the secret key. This should guarantee the safety of our fingerprint mechanism against attacks which attempt to tamper with the baseline fingerprint.

5.2 Timing Validation

Another attack strategy may be to subvert the calls to `clock_gettime`. For this, the attacker must only have knowledge of the time-based nature of our fingerprint, and not the implementation details. In this sense, an attacker with access to the program executable can identify the `clock_gettime` call-sites by inspecting the assembly:

```
1  $ objdump -d main
2
3  # wrapper around the C library clock_gettime
4  00000000000036d0 <clock_gettime@plt>:
5      36d0:   f3 0f 1e fa           endbr64
6      36d4:   f2 ff 25 65 e7 00 00  bnd jmp *0xe765(%rip) #
           11e40 <clock_gettime@GLIBC_2 .17>
7      36db:   0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)
8
9  # non-trivial wrapper around the previous wrapper
10 0000000000004446 <_Z14_clock_gettimeiP8timespec>:
11     4446:   f3 0f 1e fa           endbr64
12     444a:   55                    push   %rbp
13     444b:   48 89 e5              mov    %rsp,%rbp
14     444e:   48 83 ec 20           sub    $0x20,%rsp
15     4452:   89 7d ec              mov    %edi,-0x14(%rbp)
16     4455:   48 89 75 e0           mov    %rsi,-0x20(%rbp)
17     4459:   0f ae f0              mfence
18     445c:   90                    nop
19     445d:   48 8b 55 e0           mov    -0x20(%rbp),%rdx
20     4461:   8b 45 ec              mov    -0x14(%rbp),%eax
21     4464:   48 89 d6              mov    %rdx,%rsi
22     4467:   89 c7                 mov    %eax,%edi
23     4469:   e8 62 f2 ff ff        call   36d0
           <clock_gettime@plt>
24     446e:   89 45 fc              mov    %eax,-0x4(%rbp)
25     4471:   0f ae f0              mfence
26     4474:   90                    nop
27     4475:   8b 45 fc              mov    -0x4(%rbp),%eax
28     4478:   c9                    leave
29     4479:   c3                    ret
30
31 # fingerprint timing routine
32     4c3c:   e8 05 f8 ff ff        call   4446
           <_Z14_clock_gettimeiP8timespec>
33     ...
34     4c6b:   e8 d6 f7 ff ff        call   4446
           <_Z14_clock_gettimeiP8timespec>
```

Here, it is worth noting that line `4469` calls the C library `clock_gettime` wrapper, whose result (supposedly stored in the `eax` register), is moved onto the stack in line `446e`. If, instead of writing the contents of `eax` onto the stack, the attacker can write the value 0, they will generate a reproducible null fingerprint. One way to achieve this is to replace the instruction

```
1      446e:   89 45 fc              mov    %eax,-0x4(%rbp)
```

with

```
1      446e:   c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
```

However, the new instruction has a different width, which could modify the address of further instructions. To avoid this, the new instruction (which poisons the stack with null values) can instead replace the call into the C library's `clock_gettime`, which has the same width. Then, to avoid the null value being overwritten by a garbage value stored in `eax`,

```
    446e:   89 45 fc             mov    %eax,-0x4(%rbp)
```

can be replaced with

```
    4475:   8b 45 fc             mov    -0x4(%rbp),%eax
```

This effectively behaves as a no-op, since it is repeated a few instructions later. The result of making these two changes should be a reproducible null-fingerprint generator. To install these changes, the attacker can find the instructions of interest in the binary file:

```
$ hexedit main

1E FA 55 48 89 E5 48 83 EC 20 89 7D EC 48 89 75 E0 0F AE F0
90 48 8B 55 E0 8B 45 EC 48 89 D6 89 C7 E8 62 F2 FF FF 89 45
FC 0F AE F0 90 8B 45 FC C9 C3 F3 0F 1E FA 55 48 89 E5 53 48
83 EC 48 48 89 7D B8 48 89 75 B0 64 48 8B 04 25 28 00 00 00
...
F3 0F 1E FA F2 FF 25 75 E7 00 00 0F 1F 44 00 00 F3 0F 1E FA
F2 FF 25 6D E7 00 00 0F 1F 44 00 00 F3 0F 1E FA F2 FF 25 65
E7 00 00 0F  1F 44 00 00 F3 0F 1E FA F2 FF 25 5D E7 00 00 0F
1F 44 00 00 F3 0F 1E FA F2 FF 25 55 E7 00 00 0F 1F 44 00 00
```

and replace them with the new instructions

```
1E FA 55 48 89 E5 48 83 EC 20 89 7D EC 48 89 75 E0 0F AE F0
90 48 8B 55 E0 8B 45 EC 48 89 D6 89 C7 E8 62 F2 FF FF 8B 45
FC 0F AE F0 90 8B 45 FC C9 C3 F3 0F 1E FA 55 48 89 E5 53 48
83 EC 48 48 89 7D B8 48 89 75 B0 64 48 8B 04 25 28 00 00 00
...
F3 0F 1E FA F2 FF 25 75 E7 00 00 0F 1F 44 00 00 F3 0F 1E FA
F2 FF 25 6D E7 00 00 0F 1F 44 00 00 F3 0F 1E FA C7 45 FC 00
00 00 00 0F  1F 44 00 00 F3 0F 1E FA F2 FF 25 5D E7 00 00 0F
1F 44 00 00 F3 0F 1E FA F2 FF 25 55 E7 00 00 0F 1F 44 00 00
```

The resulting binary generates a null-fingerprint with consistent 2000/2000 match scores. Once again, the attacker required nothing more than knowledge of the time-based nature of the fingerprint. Although this particular attack could be mitigated by a simple timing validation (e.g. check that measurements are non-empty or that the entropy is above some threshold), the underlying problem is more fundamental. Namely, the implementation is vulnerable to replay attacks, where an attacker records all the system calls and their return values during the baseline fingerprint generation, and replays them to reproduce the baseline fingerprint.

5.2.1 Forced Timing Randomness. To prevent against system-call subversion or system-call replay attacks, we implement randomized timing validation, which rejects a fake stream of system-call return values with very high probability. In this sense, we intersperse the real measurements with "bait" measurements, which randomly time code-snippets of varying lengths (either 0 or 1000 instructions) depending on a source of boolean randomness called `race`:

```
    TIMING_THRESHOLD = 1000;
    violations = 0, tests = 0;
    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (!race) {
                _clock_gettime(CLOCK_REALTIME, &startTime);
                _clock_gettime(CLOCK_REALTIME, &endTime);
                delta = endTime - startTime;
                violations += (delta > TIMING_THRESHOLD);
                tests += 1;
            }
            ...
            // fingerprint measurement
            ...
            if (race) {
                _clock_gettime(CLOCK_REALTIME, &startTime);
                for (volatile int i = 1000; i > 0; i--)
                    ;
                _clock_gettime(CLOCK_REALTIME, &endTime);
                delta = endTime - startTime;
                violations += (delta < TIMING_THRESHOLD);
                tests += 1;
            }
        }
    }
    if (1.0 * violations > 0.1 * tests)
        // Attack!
```

Thus (a) the positions of the true measurements in the stream are non-deterministic (so a strict system-call replay would very likely feed "bait" values into the true measurements, invalidating the fingerprint) and (b) without prior knowledge of (or control over) the randomness, the attacker cannot match the "bait" measurements with the length of their respective code-snippets, making the attack detectable. In this sense, we detect an attack if more than 10% of the "bait" measurements are wrong. This should allow for context-switch related noise.

Additional consideration must be given to the source of randomness. As an attacker with control over all system-calls can also control randomness obtained from trapping into the kernel, it is preferable to implement a userspace source of randomness. We do this by scheduling a concurrent thread, which continuously modifies the `race` variable:

```
volatile bool race = false;
std::atomic<bool> t = true;
void* add(void* /*unused*/)
{
    while (t)
        race = !race;
    return NULL;
}
```

Therefore, the source of randomness lies in the thread scheduling and instruction execution order, which is more difficult for the attacker to control.

6 RESULTS

[Sanchez-Rola et al. 2018] defined six key characteristics by which to assess the strength of a fingerprinting scheme: discrimination power, stability, homogeneous discrimination, efficiency, resilience to evasion, and resilience to change. Discrimination power is a statistical metric quantifying the rate at which the fingerprinting algorithm produces distinct fingerprints for different machines, while homogeneous discrimination concerns the special case of whether the method still produces unique fingerprints for machines in the same "homogeneous family" that share identical hardware and software components. Stability is the other key numerical property and is calculated as the percentage of the time that the algorithm produces the same fingerprint for a single machine across multiple measurements, while resilience to change extends the concept of stability across a longer period of time and checks whether the fingerprint remains stable across an extended duration as the system's hardware ages. Finally, efficiency measures the time needed to generate fingerprints or test whether a machine matches an existing fingerprint template, and resilience to evasion describes the capability of the technique to resist any adversarial attempts to tamper with the fingerprint generation or circumvent the comparison process.

In this section we will evaluate the results of running our implementation of CryptoFP on our virtual machine testing environment in order to identify the stability, discrimination power (particularly in homogeneous families), and efficiency of our method on standardized hardware to demonstrate its strength as both a fingerprinting and license locking technique and highlight the benefits of our modifications compared to the original implementation of CryptoFP. We will also briefly comment on the resilience to change, though acknowledge that the limited timespan of this study makes it difficult to fully predict the effects of clock crystal aging over the entire lifespan of a system. Our discussion of resilience to evasion is covered in Section 5 through our analysis of the ability to resist attacks that could affect the fingerprint's stability or discrimination power, and by extension violate the security of our license locking system.

6.1 Testing Environment and Automation

The original native implementation of CryptoFP was evaluated on two sets of homogeneous host machines (containing 89 and 176 computers, respectively), providing a robust platform for testing the stability and discrimination power of the algorithm, particularly within the context of homogeneous discrimination. While we did not have access to a large number of identical host machines for these experiments, we chose to replicate the creation of a homogeneous family through the use of cloud computing platforms by generating virtual machine instances with identical hardware and software configurations. The initial prototyping phase of our study utilized Google Cloud Platform Compute Engine VMs due to the ease of setup and flexibility in generating machines with a wide variety of hardware configurations (including different virtual CPU types and core counts).

However, the preliminary analysis from running our first version of CryptoFP on the GCP test suite indicated that the TSC-based approach described in Section 4 could consistently distinguish between machines with distinct hardware but was incapable of differentiating between homogeneous machines that shared the same hardware specifications. This result is reasonable considering that the TSC counts the total number of elapsed CPU cycles on a chip and therefore can be expected to measure the same amount of elapsed time when executing the fingerprinting function across any group of machines that share the same CPU or vCPU model. While this result prompted our development of a new adaptation of CryptoFP that utilized an alternative clocksource, it also required us to move away from GCP as our virtual machine provider. Our analysis of the kernel (described in detail in Appendix A) revealed that the KVM hypervisor used by GCP obscures many hardware details related to timing and only exposes access to the TSC and KVM_CLOCK sources, with the latter being a paravirtualized clock derived directly from the TSC and therefore offering no additional capability for homogeneous discrimination.

While GCP virtual machines do not allow sufficient access to the underlying hardware clocksources, Amazon Web Services EC2 VMs provide a more realistic simulation of a host machine including granting direct access to the HPET timer with *sudo* permissions, allowing for us to run our second iteration of CryptoFP. Our final test suite included 60 t2.micro VM instances each running Ubuntu 24.04, using this single homogeneous group to isolate the unique variations between each machine's clock crystals rather than the macroscopic differences between hardware. The size of this test suite required us to develop an automated script to perform all of the necessary setup and run the fingerprint comparisons on every machine in order to compute the overall discrimination power and stability for the homogeneous family. This script was implemented in Python and took advantage of the Paramiko library to establish SSH connections with each of the machines from a central host, install all of the required packages and build the fingerprinting software on each new VM, generate a fingerprint for each machine, distribute all 60 fingerprints to each of the 60 VMs over SCP, and then use multithreading to simultaneously run the comparison algorithm on every machine with each of the fingerprints. The comparison with the fingerprint originally generated on that machine (e.g. Fingerprint #30 on VM #30) was used to track the overall stability, while the comparisons with the other 59 fingerprints were used to check for homogeneous discrimination power.

6.2 Efficiency

Before entering into the discussion of the stability and discrimination results, we will first briefly cover our recorded efficiency metrics. For the sake of consistency, all timing information was recorded on the t2.micro VMs in our standardized EC2 test suite, which contain a single core 3.3 GHz vCPU, though it is evident that these values will differ between hardware and will likely be faster on a dedicated host with a modern multi-core CPU. Ultimately, the difference between our TSC-based and HPET-based implementations was found to be negligible, with the additional two read operations of the TSC in the first approach adding less than 10 milliseconds of overhead. In both cases, the fingerprint generation process completed in an average of **2.35 seconds** across a complete test batch for the default algorithm, though activating the encryption strategy

discussed in Section 5.1 causes the duration to rise to over 4 minutes in order to fully generate and encrypt the fingerprint. In terms of the comparison, running an individual, unencrypted fingerprint check took an average of just **0.285 seconds** to complete, highlighting the efficiency of the CryptoFP approach compared to other competing fingerprint algorithms. It must be noted that when running the full testing script the average fingerprint comparison time did increase (ranging from **2.86s**-**3.39s**), though analysis of the CPU utilization charts indicated that EC2 was throttling CPU to a limit of just 10-13% after an initial spike at the start of the experiment as seen in Fig 1. However, this situation of running 360,000 fingerprint comparisons on a single machine would never be replicated in practice during the execution of the license locking software, so it can be safely assumed that the small batch average of **0.285s** is an appropriate standard for the efficiency of the algorithm. In contrast, encrypted fingerprint comparisons have been recorded to take approximately 10 seconds on t2.micro machines, which is still reasonably efficient within the context of license locking given that this step will occur during a software installation process.

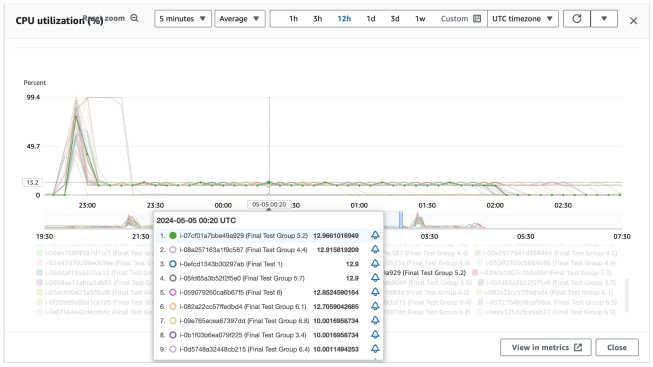

Fig. 1. Graph of CPU utilization for all 60 t2.micro VMs showing the CPUs being throttled to 10-13% of the maximum, accounting for the increase in execution time.

6.3 Discrimination Power

While the efficiency results demonstrated the potential of our CryptoFP implementation as a practical fingerprinting strategy, achieving a high discrimination power is imperative to enabling the fingerprint to serve as a valid license locking mechanism; this is due to the fact that a system with low discrimination power will frequently return that a given machine matches a different machine's fingerprint, which could allow for the unauthorized sharing of a license locked binary. This requirement was the main motivation behind our shift from the TSC-based approach to the HPET-based approach, as the TSC version achieved 100% discrimination power between heterogeneous machines but was incapable of ever distinguishing between machines in the same homogeneous family and therefore recorded 0% homogeneous discrimination. Although real-world host devices evidently display more hardware diversity than the single homogeneous family of our EC2 test suite, it is important to recognize that many consumer devices are homogeneous (consider for example all base versions of a specific laptop release in the same model year), so it is critical that our method is able to handle homogeneity.

Our HPET implementation also achieved 100% heterogeneous discrimination amongst all unique machines that we tested it on, but it also demonstrated the ability to distinguish between homogeneous machines in our early tests during development. In order to fully test the homogeneous discrimination power of the HPET implementation we employed our automated testing script described in Section 6.1, which generates a fingerprint for all 60 EC2 virtual machines then runs a comparison with each fingerprint on every machine to measure the ability to discriminate between these homogeneous systems based upon their clock imperfections. The relevant data for computing the discrimination power concerns all of the comparisons with the 59 fingerprints generated on different machines, giving a total of **354,000 comparisons** in each execution of our testing script. Despite this incredibly large sample size, we never observed a single "collision" in which CryptoFP returned a positive match result for a fingerprint generated on a different VM, giving an **overall discrimination power and homogeneous discrimination power of 100%**.

6.4 Stability and Resilience to Change

Our automated testing script also computes stability results within the same pass as the discrimination power, but focuses solely on each virtual machine's single comparison to the fingerprint generated on that VM. With a test suite of 60 VMs and running each fingerprint 100 times, this gives a sample size of **6000 comparisons** for each pass of the testing script. We focused our stability tests solely on the HPET implementation due to the lack of homogeneous discrimination power in the TSC version preventing it from serving as a viable method for license locking. For our stability tests we also wanted to ensure that our fingerprinting method was stable under nonstandard conditions, so we added a feature to our testing script that applied varying stress to the system during testing. This allowed us to measure whether the fingerprints were still reproducible even when the system was under 100% CPU load, and when the resulting increase in CPU temperature caused the clock crystals to oscillate faster. Our final results reflected that fingerprints were in fact stable overall and there was little drop-off in stability when applying this varying load, with the system achieving **97.72%** stability under no load and **96.32%** stability under load. While this does reveal that there is a small drop in stability under peak load, the effect is negligible enough that it does not risk compromising the functionality of the license locking system. Our results have reflected that individual fingerprint comparisons are approximately independent of each other and as a result it is possible to achieve a higher stability result by sequentially taking several fingerprints and accepting a match if any one of the comparisons matches; this has been shown to yield an overall stability of at least **99.5%** when performing at most 5 comparisons, while still maintaining efficiency with a total runtime of less than 2 seconds to complete the full test batch.

7 ALTERNATIVE FINGERPRINT METHODS

In this section, we discuss an alternative fingerprinting method to CryptoFP. By using two threads on separate cores to write to the same memory region simultaneously, it is possible to generate a byte sequence that could potentially be used as a fingerprint.

In the common case, we prevent multiple threads from writing to the same memory simultaneously to ensure data consistency and prevent data corruption. Because under an idealistic situation, when 2 cores write different data to the same memory location at the same time, the behavior is undefined and either data is possible to be the final result. Therefore, when we use 2 threads to write consistent 0s or 1s to the same memory simultaneously, the result is highly likely to be a random sequence of 0s and 1s. In the real scenario, however, due to the unique imperfection of each core during manufacturing progress, although threads write simultaneously, tiny time differences can convert a random output to a patterned result. Therefore, we take advantage of this variation between cores to perform a shared array write to generate a sequence that could possibly be used for fingerprinting. To achieve this, we

(1) declare a global byte array as the target memory location.
(2) create 2 threads that write 0's or 1's to the target memory separately.
(3) pin these threads to 2 fixed cores to make sure the program always uses the same differences of 2 cores.
(4) make 2 threads start at exactly the same time so that 2 threads write the same position of the array simultaneously.
(5) get the value of the byte array to use as a possible fingerprint.

If our hypothesis is correct, we can retrieve a valid patterned 01 sequence instead of an array with all 1s or all 0s or a totally random series when both threads start simultaneously.

Step (3) is extremely important because the variation between 2 cores is tiny. The synchronization must be very accurate to retrieve such differences in our results. Also, a slight delay of a later thread will cause the result of the former thread to be overwritten, making the final result all 0s or all 1s. Therefore, we need to use some synchronization method to ensure 2 threads start at the same time in a tick level. At first, we try to use pthread_barrier as the synchronization function, however the average start time difference between 2 threads is more than 10000 ticks. Then, instead of a complicated function call, we simply use a global variable that marks whether threads should start. Two threads just keep spinning in the loop before the sign is set. By using this method, we successfully reduced the start time difference to less than 200 ticks. To get higher accuracy, we add a loop that keeps fingerprinting until the time difference is less than a threshold, and then we save the result as the final output. Since one fingerprinting operation is very fast, repeating this still takes a short time. As a result, it takes an average of less than 3 seconds to get a time difference of less than 10 ticks.

We test this method within different start time difference ranges. When the difference is larger than 100 ticks, most of the outputs are all 1s or all 0s since the later thread is way behind the earlier one. When the difference lies in 30 to 100 ticks, small sets of continuous 1s appear in the array of 0s or vice versa. However, the lengths of these small sets vary each time we fingerprint, and they do not show any obvious pattern. When the difference is less than 30 ticks, including 0 tick, all 1s or all 0s outputs take the dominant place again, and small sets of the other value appear less frequently. In conclusion, instead of getting valid patterned 01 outputs when the start time difference reaches 0, we observe valid but random outputs when the difference is between 30 to 100 ticks.

Our observation is unpromising for several reasons. One reason is the inaccuracy of our time measurement method. We use *rdtsc* instruction in the thread function to get the actual start time. However, this instruction takes about 25-35 cycles, which is not negligible in our experiment. Therefore, we are still unsure whether 2 threads start at the same tick. Another reason is that the variable CPU states, rather than the tiny differences between cores, determine the final output. Although 2 threads take the same length of instructions to execute, the actual execution time is also determined by CPU frequency, which is different between cores and varies over time. And the execution time difference caused by this is much larger than that caused by core imperfections. Thus, the output array can not actually reflect the small variation between cores.

In conclusion, the fingerprinting method, which takes advantage of tiny variations of cores, is theoretically feasible but practically challenging due to the difficulty of thread synchronization and variable CPU states. However, it is still worth further discovering similar methods by using unique imperfections of hardware.

8 FUTURE WORK

Future improvements to our fingerprint-based license-locking mechanism include (a) identifying two collocated clocksources available from userspace (b) performing better code obfuscation to prevent against more complex attacks (which could attempt to reverse engineer our obfuscated executable) and (c) faster functional encryption procedures to reduce fingerprinting time.

REFERENCES

2021. CiFEr. https://github.com/fentec-project/CiFEr?tab=readme-ov-file. [Online; accessed 27-April-2024].

2024. FingerprintJS. https://github.com/fingerprintjs/fingerprintjs. [Online; accessed 27-April-2024].

Michel Abdalla, Bourse Florian, Angelo Caro, and David Pointcheval. 2015. Simple Functional Encryption Schemes for Inner Products. https://doi.org/10.1007/978-3-662-46447-2_33

Alexandr Andoni. 2021a. Hashing. https://www.cs.columbia.edu/~andoni/advancedS21/scribes/scribe3.pdf. [Online; accessed 27-April-2024].

Alexandr Andoni. 2021b. Hashing. https://www.cs.columbia.edu/~andoni/advancedS21/scribes/scribe4.pdf. [Online; accessed 27-April-2024].

Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. *SIGARCH Comput. Archit. News* 42, 3 (jun 2014), 361–372. https://doi.org/10.1145/2678373.2665726

Timothy Salo. 2007. MultiFactor Fingerprints for Personal Computer Hardware. (10 2007). https://doi.org/10.1109/MILCOM.2007.4455113

Iskander Sanchez-Rola, Igor Santos, and Davide Balzarotti. 2018. Clock Around the Clock: Time-Based Device Fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1502–1514. https://doi.org/10.1145/3243734.3243796

Hari Venugopalan, Kaustav Goswami, Zainul Abi Din, Jason Lowe-Power, Samuel T. King, and Zubair Shafiq. 2023. Centauri: Practical Rowhammer Fingerprinting. arXiv:2307.00143 [cs.CR]

Mark N. Wegman and J.Lawrence Carter. 1981. New hash functions and their use in authentication and set equality. *J. Comput. System Sci.* 22, 3 (1981), 265–279. https://doi.org/10.1016/0022-0000(81)90033-7

A TOP-DOWN OVERVIEW OF UNIX CLOCKS

In this appendix, we argue that calling `clock_gettime` with a `clockid_t` value in {`CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW, CLOCK_PROCESS_CPUTIME_ID`} will likely return a TSC-based timestamp on x86 architectures. Although not explored here explicitly, `CLOCK_BOOTTIME`, `CLOCK_TIME` and `CLOCK_THREAD_CPUTIME_ID` indirectly rely on the same clock [2].

A.1 Trapping Into the Kernel

The `<time.h> int clock_gettime(clockid_t, struct timespec *)`[3] takes in a `clockid_t` (e.g. `CLOCK_REALTIME, CLOCK_MONOTONIC_RAW`) and reads the current time (in seconds and nanoseconds) into the `struct timespec` passed as an argument:

```
int __clock_gettime(clockid_t clk, struct timespec *ts)
{
   int r;
#ifdef SYS_clock_gettime64
   r = -ENOSYS;
   if (sizeof(time_t) > 4)
      r = __syscall(SYS_clock_gettime64, clk, ts);
   if (SYS_clock_gettime == SYS_clock_gettime64 ||
        r!=-ENOSYS)
      return __syscall_ret(r);
   long ts32[2];
   r = __syscall(SYS_clock_gettime, clk, ts32);
#endif
   ...
}
```

This is a thin wrapper around the `SYS_clock_gettime` system call[4]

```
SYSCALL_DEFINE2(clock_gettime, constclockid_t, which_clock,
     structtimespec__user *,tp)
{
   structk_clock *kc = clockid_to_kclock(which_clock);
   structtimespec kernel_tp;
   int error;
   if (!kc)
      return -EINVAL;
   error = kc->clock_get(which_clock, &kernel_tp);
   if (!error &&copy_to_user(tp, &kernel_tp, sizeof
        (kernel_tp)))
      error = -EFAULT;
   return error;
}
```

which gets the reference to the desired clock via `clockid_to_kclock`[5]:

```
static structk_clock *clockid_to_kclock(constclockid_t id)
{
   ...
   return &posix_clocks[id];
}
```

[2] timekeeping.c on Bootlin Elixir
[3] clock_gettime.c
[4] posix-timers.c on Bootlin Elixir
[5] posix-timers.c on Bootlin Elixir

A.2 POSIX Timers

These references are created by `posix_timers_register_clock`[6]

```
void posix_timers_register_clock(constclockid_tclock_id,
     structk_clock *new_clock)
{
   ...
   posix_clocks[clock_id] = *new_clock;
}
```

invoked on startup by the function `init_posix_timers` (in the case of `CLOCK_RUNTIME, CLOCK_MONOTONIC` and `CLOCK_MONOTONIC_RAW`) [7]

```
static__init int init_posix_timers(void)
{
   structk_clockclock_realtime = {
      .clock_getres =posix_get_hrtimer_res,
      .clock_get =posix_clock_realtime_get,
      .clock_set =posix_clock_realtime_set,
      ...
   };
   structk_clockclock_monotonic = {
      .clock_getres =posix_get_hrtimer_res,
      .clock_get =posix_ktime_get_ts,
      ...
   };
   structk_clockclock_monotonic_raw = {
      .clock_getres =posix_get_hrtimer_res,
      .clock_get =posix_get_monotonic_raw,
   };
   ...
   posix_timers_register_clock(CLOCK_REALTIME,
        &clock_realtime);
   posix_timers_register_clock(CLOCK_MONOTONIC,
        &clock_monotonic);
   posix_timers_register_clock(CLOCK_MONOTONIC_RAW,
        &clock_monotonic_raw);
    ...
}
```

and by `init_posix_cpu_timers` (for `CPU_PROCESS_CPUTIME_ID`) [8]:

```
static __init int init_posix_cpu_timers(void)
{
   struct k_clock process = {
      .clock_getres = process_cpu_clock_getres,
      .clock_get = process_cpu_clock_get,
      .timer_create = process_cpu_timer_create,
      .nsleep    = process_cpu_nsleep,
      .nsleep_restart = process_cpu_nsleep_restart,
   };
   struct k_clock thread = {
      .clock_getres = thread_cpu_clock_getres,
      .clock_get = thread_cpu_clock_get,
      .timer_create = thread_cpu_timer_create,
   };
   struct timespec ts;
```

[6] posix-timers.c on Bootlin Elixir
[7] posix-timers.c on Bootlin Elixir
[8] posix-cpu-timers.c on Bootlin Elixir

```
16
17    posix_timers_register_clock(CLOCK_PROCESS_CPUTIME_ID,
          &process);
18    posix_timers_register_clock(CLOCK_THREAD_CPUTIME_ID,
          &thread);
19     ...
20 }
```

The syscall then uses the getter `kc->clock_get()`[9].

The `CLOCK_REALTIME` and `CLOCK_MONOTONIC` clocks are identical up to shifting by the `tk->wall_to_monotonic` offset. These two are adjusted by NTP, whereas `CLOCK_MONOTONIC_RAW` is not affected by NTP. To our understanding, all the seqlocks used here and in the upcoming sections are just an artifact of how synchronization is performed in the kernel, and so are not relevant for this timekeeping inquiry.

```
1  /* Get clock_realtime */
2  static int posix_clock_realtime_get(clockid_t which_clock,
       structtimespec *tp)
3  {
4     ktime_get_real_ts(tp);
5     return 0;
6  }
7  static inline void ktime_get_real_ts(struct timespec *ts)
8  {
9     getnstimeofday64(ts);
10 }
11 /**
12  * __getnstimeofday64 - Returns the time of day in a
        timespec64.
13  * @ts:    pointer to the timespec to be set
14  *
15  * Updates the time of day in the timespec.
16  * Returns 0 on success, or -ve when suspended (timespec
        will be undefined).
17  */
18 int __getnstimeofday64(struct timespec64 *ts)
19 {
20    struct timekeeper *tk = &tk_core.timekeeper;
21    unsigned long seq;
22    s64 nsecs = 0;
23
24    do {
25       seq = read_seqcount_begin(&tk_core.seq);
26       ts->tv_sec = tk->xtime_sec;
27       nsecs = timekeeping_get_ns(&tk->tkr_mono);
28
29    } while (read_seqcount_retry(&tk_core.seq, seq));
30
31    ts->tv_nsec = 0;
32    timespec64_add_ns(ts, nsecs);
33     ...
34 }
```

[9]By following the clock setup above, the getters of interest are

(1) posix-timers.c and timekeeping.c for `CLOCK_REALTIME`
(2) posix-timers.c and timekeeping.c for `CLOCK_MONOTONIC`
(3) posix-timers.c and timekeeping.c for `CLOCK_MONOTONIC_RAW`
(4) posix-cpu-timers.c, posix-cpu-timers.c, posix-cpu-timers.c and posix-cpu-timers.c for `CLOCK_CPUTIME_ID`

```
1  /*
2   * Get monotonic time for posix timers
3   */
4  static int posix_ktime_get_ts(clockid_t which_clock, struct
       timespec *tp)
5  {
6     ktime_get_ts(tp);
7     return 0;
8  }
9  static inline void ktime_get_ts(struct timespec *ts)
10 {
11    ktime_get_ts64(ts);
12 }
13 /**
14  * ktime_get_ts64 - get the monotonic clock in timespec64
        format
15  * @ts:    pointer to timespec variable
16  *
17  * The function calculates the monotonic clock from the
        realtime
18  * clock and the wall_to_monotonic offset and stores the
        result
19  * in normalized timespec64 format in the variable pointed
        to by @ts.
20  */
21 void ktime_get_ts64(struct timespec64 *ts)
22 {
23    struct timekeeper *tk = &tk_core.timekeeper;
24    struct timespec64 tomono;
25    s64 nsec;
26    unsigned int seq;
27     ...
28    do {
29       seq = read_seqcount_begin(&tk_core.seq);
30       ts->tv_sec = tk->xtime_sec;
31       nsec = timekeeping_get_ns(&tk->tkr_mono);
32       tomono = tk->wall_to_monotonic;
33
34    } while (read_seqcount_retry(&tk_core.seq, seq));
35
36    ts->tv_sec += tomono.tv_sec;
37    ts->tv_nsec = 0;
38    timespec64_add_ns(ts, nsec + tomono.tv_nsec);
39 }
```

```
1  /*
2   * Get monotonic-raw time for posix timers
3   */
4  static int posix_get_monotonic_raw(clockid_t which_clock,
       struct timespec *tp)
5  {
6     getrawmonotonic(tp);
7     return 0;
8  }
9  static inline void getrawmonotonic(struct timespec *ts)
10 {
11    getrawmonotonic64(ts);
12 }
13
```

```c
/**
 * getrawmonotonic64 - Returns the raw monotonic time in a timespec
 * @ts:    pointer to the timespec64 to be set
 *
 * Returns the raw monotonic time (completely un-modified by ntp)
 */
void getrawmonotonic64(struct timespec64 *ts)
{
   struct timekeeper *tk = &tk_core.timekeeper;
   struct timespec64 ts64;
   unsigned long seq;
   s64 nsecs;

   do {
      seq = read_seqcount_begin(&tk_core.seq);
      nsecs = timekeeping_get_ns(&tk->tkr_raw);
      ts64 = tk->raw_time;

   } while (read_seqcount_retry(&tk_core.seq, seq));

   timespec64_add_ns(&ts64, nsecs);
   *ts = ts64;
}
```

```c
#define PROCESS_CLOCK MAKE_PROCESS_CPUCLOCK(0, CPUCLOCK_SCHED)
static int process_cpu_clock_get(const clockid_t which_clock, struct timespec *tp)
{
   return posix_cpu_clock_get(PROCESS_CLOCK, tp);
}

static int posix_cpu_clock_get(const clockid_t which_clock, struct timespec *tp)
{
   const pid_t pid = CPUCLOCK_PID(which_clock);
   int err = -EINVAL;

   if (pid == 0) {
      /*
       * Special case constant value for our own clocks.
       * We don't have to do any lookup to find ourselves.
       */
      err = posix_cpu_clock_get_task(current, which_clock, tp);
   } else { ... }

   return err;
}

static int posix_cpu_clock_get_task(struct task_struct *tsk,
          const clockid_t which_clock, struct timespec *tp)
{
   int err = -EINVAL;
   unsigned long long rtn;

   if (CPUCLOCK_PERTHREAD(which_clock)) { ... }
   else {
      if (tsk == current || thread_group_leader(tsk))
         err = cpu_clock_sample_group(which_clock, tsk, &rtn);
   }

   if (!err)
      sample_to_timespec(which_clock, rtn, tp);
   return err;
}

/*
 * Sample a process (thread group) clock for the given group_leader task.
 * Must be called with task sighand lock held for safe while_each_thread()
 * traversal.
 */
static int cpu_clock_sample_group(const clockid_t which_clock,
             struct task_struct *p,
             unsigned long long *sample)
{
   struct task_cputime cputime;
   switch (CPUCLOCK_WHICH(which_clock)) {
      ...
   case CPUCLOCK_SCHED:
      thread_group_cputime(p, &cputime);
      *sample = cputime.sum_exec_runtime;
      break;
   }
   return 0;
}
```

A.3 REALTIME/MONOTONIC (RAW): Struct Timekeeper

The main timekeeping data structure is `struct timekeeper`, which stores pointers to all the measurements used/mentioned in the previous sections/functions[10]

```c
/**
 * struct timekeeper - Structure holding internal timekeeping values.
 * @tkr_mono: readout base structure for CLOCK_MONOTONIC
 * @tkr_raw:  readout base structure for CLOCK_MONOTONIC_RAW
 * @xtime_sec: Current CLOCK_REALTIME time in seconds
 * @ktime_sec: Current CLOCK_MONOTONIC time in seconds
 * @wall_to_monotonic: CLOCK_REALTIME to CLOCK_MONOTONIC offset
 * @offs_real: Offset clock monotonic -> clock realtime
 * @offs_boot: Offset clock monotonic -> clock boottime
 * @offs_tai: Offset clock monotonic -> clock tai
 * @tai_offset: The current UTC to TAI offset in seconds
 * @clock_was_set_seq: The sequence number of clock was set events
 * @next_leap_ktime: CLOCK_MONOTONIC time value of a pending leap-second
 * @raw_time: Monotonic raw base time in timespec64 format
 * ...
 */
```

[10] timekeeper_internal.h on Bootlin Elixir

```
struct timekeeper {
   struct tk_read_base tkr_mono;
   struct tk_read_base tkr_raw;
   u64        xtime_sec;
   unsigned long   ktime_sec;
   struct timespec64 wall_to_monotonic;
   ktime_t       offs_real;
   ktime_t       offs_boot;
   ktime_t       offs_tai;
   s32        tai_offset;
   unsigned int    clock_was_set_seq;
   ktime_t       next_leap_ktime;
   struct timespec64 raw_time;
    ...
};
```

The two members of `struct timekeeper`, `tkr_mono` and `tkr_raw`, accessed by the three clocks (`CLOCK_REALTIME`, `CLOCK_MONOTONIC` and `CLOCK_MONOTONIC_RAW`), are defined as `struct tk_read_bases`[11]:

```
/**
 * struct tk_read_base - base structure for timekeeping
     readout
 * @clock: Current clocksource used for timekeeping.
 * @read: Read function of @clock
 * @mask: Bitmask for 2's complement sub of non 64b clocks
 * @cycle_last: @clock cycle value at last update
 * @mult: multiplier for scaled math conversion
 * @shift: Shift value for scaled math conversion
 * @xtime_nsec: Shifted nano seconds offset for readout
 * @base: ktime_t (nanoseconds) base time for readout
 * ...
 */
struct tk_read_base {
   struct clocksource *clock;
   cycle_t       (*read)(struct clocksource *cs);
   cycle_t       mask;
   cycle_t       cycle_last;
   u32        mult;
   u32        shift;
   u64        xtime_nsec;
   ktime_t       base;
};
```

The `struct tk_read_base` is then accessed by the three getters via the `timekeeping_get_ns()` function[12]. Underneath, this function collects the `timekeeping_get_delta()`[13] and performs a multiply and shift operation on it.

To our understanding, a `struct tk_read_base` keeps a reference to a clock (whose details, e.g. increments/updates, are abstracted away), the value of that clock at the last update, and the true time at that same last update (pre-shift). To get the time from the `struct tk_read_base`, the clock-delta `current_time - last_update_time` is converted from ticks to the same units as the true time (via the `tkr->mult` field), then added onto the `last_update` true time. The `tkr->shift` field then controls the resolution of our clock.

[11] timekeeper_internal.h on Bootlin Elixir
[12] timekeeping.c on Bootlin Elixir
[13] timekeeping.c and timekeeping_internal.h on Bootlin Elixir

Rephrased, the time of this clock is the time at the previous update, plus the internal-clock delta since that previous update, converted to common units. The result is shifted to the desired resolution.

```
s64 timekeeping_get_ns(struct tk_read_base *tkr)
{
   cycle_t delta;
   s64 nsec;

   delta = timekeeping_get_delta(tkr);
   nsec = delta * tkr->mult + tkr->xtime_nsec;
   nsec >>= tkr->shift;

   /* If arch requires, add in get_arch_timeoffset() */
   return nsec + arch_gettimeoffset();
}

cycle_t timekeeping_get_delta(struct tk_read_base *tkr)
{
   struct timekeeper *tk = &tk_core.timekeeper;
   cycle_t now, last, mask, max, delta;
   unsigned int seq;

   do {
      seq = read_seqcount_begin(&tk_core.seq);
      now = tkr->read(tkr->clock);
      last = tkr->cycle_last;
      mask = tkr->mask;
   } while (read_seqcount_retry(&tk_core.seq, seq));
   delta = clocksource_delta(now, last, mask);
   // Handle underflow and overflow
   ...
   return delta;
}

static inline cycle_t clocksource_delta(cycle_t now,
     cycle_t last, cycle_t mask)
{
   return (now - last) & mask;
}
```

The `tk_core.timekeeper` is initialized in `timekeeping_init` [14] which is a wrapper around `tk_setup_internals` [15]. What is worth noting is that `tk->tkr_mono.clock` and `tk->tkr_raw.clock` both end up referencing `clocksource_default_clock()`. In other words, this is the primary time source in this implementation.

```
/*
 * timekeeping_init - Initializes the clocksource and
      common timekeeping values
 */
void __init timekeeping_init(void)
{
   struct timekeeper *tk = &tk_core.timekeeper;
   struct clocksource *clock;
   ...
   clock = clocksource_default_clock();
   if (clock->enable)
```

[14] timekeeping.c on Bootlin Elixir
[15] timekeeping.c on Bootlin Elixir

```
      clock->enable(clock);
   tk_setup_internals(tk, clock);
   ...
   raw_spin_unlock_irqrestore(&timekeeper_lock, flags);
}

/**
 * tk_setup_internals - Set up internals to use clocksource clock.
 * @tk:    The target timekeeper to setup.
 * @clock:    Pointer to clocksource.
 * Calculates a fixed cycle/nsec interval for a given
 * clocksource/adjustment pair and interval request.
 */
static void tk_setup_internals(struct timekeeper *tk, struct clocksource *clock)
{
   struct clocksource *old_clock;

   old_clock = tk->tkr_mono.clock;
   tk->tkr_mono.clock = clock;
   tk->tkr_mono.read = clock->read;
   tk->tkr_mono.mask = clock->mask;
   tk->tkr_mono.cycle_last = tk->tkr_mono.read(clock);

   tk->tkr_raw.clock = clock;
   tk->tkr_raw.read = clock->read;
   tk->tkr_raw.mask = clock->mask;
   tk->tkr_raw.cycle_last = tk->tkr_mono.cycle_last;

   // NTP configuration
   ...
}
```

A.4 REALTIME/MONOTONIC (RAW): Clocksources

The `clocksource_default_clock` is based on Jiffies[16]. This is the default choice of time source in the Linux kernel. Note that Jiffies is initialized with a rating of 1, which is the lowest valid rating assignable to a timesource. The clocksource used in the end is the one with highest rating.

```
struct clocksource * __init __weak clocksource_default_clock(void)
{
   return &clocksource_jiffies;
}

static struct clocksource clocksource_jiffies = {
   .name   = "jiffies",
   .rating    = 1, /* lowest valid rating*/
   .read   = jiffies_read,
   .mask   = 0xffffffff, /*32bits*/
   .mult   = NSEC_PER_JIFFY << JIFFIES_SHIFT,
   .shift     = JIFFIES_SHIFT,
   .max_cycles = 10,
};
```

[16] jiffies.c and jiffies.c on Bootlin Elixir

However, the clocksource used within `tk_core.timekeeper` can (and very likely will) be overwritten via the `change_clocksource` call[17]. That is to say the default clocksource is important, but unlikely to be used for timekeeping purposes after boot has finished and other, higher-score clocksources have been initialized.

`change_clocksource` is ultimately invoked by `clocksource_select`[18]. Lastly, this is invoked from `__clocksource_register_scale`[19] which is where new clocksources pass through at registration in the kernel.

```
/**
 * change_clocksource - Swaps clocksources if a new one is available; accumulates current time interval and initializes new clocksource
 */
static int change_clocksource(void *data)
{
   struct timekeeper *tk = &tk_core.timekeeper;
   struct clocksource *new, *old;
   unsigned long flags;

   new = (struct clocksource *) data;

   raw_spin_lock_irqsave(&timekeeper_lock, flags);
   write_seqcount_begin(&tk_core.seq);

   timekeeping_forward_now(tk);
   /*
    * If the cs is in module, get a module reference. Succeeds
    * for built-in code (owner == NULL) as well.
    */
   if (try_module_get(new->owner)) {
      if (!new->enable || new->enable(new) == 0) {
         old = tk->tkr_mono.clock;
         tk_setup_internals(tk, new);
         if (old->disable)
            old->disable(old);
         module_put(old->owner);
      } else {
         module_put(new->owner);
      }
   }
   timekeeping_update(tk, TK_CLEAR_NTP | TK_MIRROR | TK_CLOCK_WAS_SET);

   write_seqcount_end(&tk_core.seq);
   raw_spin_unlock_irqrestore(&timekeeper_lock, flags);

   return 0;
}
```

[17] timekeeping.c on Bootlin Elixir

[18] clocksource.c, clocksource.c, clocksource.c, timekeeping.c, stop_machine.c (these links are a depth-first view of the call stack leading up to the invokation of `change_clocksource`, starting from `clocksource_select`).

[19] clocksource.c on Bootlin Elixir

The Linux Kernel exposes three functions to register a new clocksource: `__clocksource_register`[20], `clocksource_register_hz`[21] and `clocksource_register_khz`[22].

The first of them is used solely by the Jiffies clocksource (which is the default clocksource anyway, as per the last section) [23].

The second of them does not register clocks at a higher rating than the TSC clocksource on x86 (see next section). It is used by the HPET clock (if available), which registers itself with a 250 rating, making it lower priority than the TSC[24].

Worth noting is that the KVM clock also registers itself via this call[25] with a rating of 400. This makes it the highest-rated clocksource when it is used. However, this clock is used primarily for virtualization, and so falls beyond our fingerprinting attempts.

```
static inline int __clocksource_register(struct clocksource
    *cs)
{
   return __clocksource_register_scale(cs, 1, 0);
}

static inline int clocksource_register_hz(struct
    clocksource *cs, u32 hz)
{
   return __clocksource_register_scale(cs, 1, hz);
}

static inline int clocksource_register_khz(struct
    clocksource *cs, u32 khz)
{
   return __clocksource_register_scale(cs, 1000, khz);
}

static struct clocksource clocksource_hpet = {
   .name   = "hpet",
   .rating    = 250,
   .read   = read_hpet,
   .mask   = HPET_MASK,
   .flags     = CLOCK_SOURCE_IS_CONTINUOUS,
   .resume    = hpet_resume_counter,
   .archdata  = { .vclock_mode = VCLOCK_HPET },
};

static struct clocksource kvm_clock = {
   .name = "kvm-clock",
   .read = kvm_clock_get_cycles,
   .rating = 400,
   .mask = CLOCKSOURCE_MASK(64),
   .flags = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

The third registration method (`clocksource_register_khz`) is used by the TSC counter (supposedly at or around boot time), which involves a quick calibration[26]. This call schedules a delayed, longer calibration of a refined TSC clocksource (`tsc_refine_calibration_work`) via the `tsc_irqwork` variable[27] which might or might not take effect, depending on whether the calibration results align with the quick boot-time TSC calibration. Nevertheless, the TSC-based clocksource is registered this way. The definition of the TSC clocksource is here.

The `clocksource_tsc` is initialized with a rating of 300 and, on a system which increments the TSC counter at a fixed rate, which includes most of the latest systems, this rating is unlikely to be downgraded. This leads us to the conclusion that all of `CLOCK_REALTIME`, `CLOCK_MONOTONIC` and `CLOCK_MONOTONIC_RAW` are based on the TSC counter.

```
static struct clocksource clocksource_tsc = {
   .name                = "tsc",
   .rating              = 300,
   .read                = read_tsc,
   .mask                = CLOCKSOURCE_MASK(64),
   .flags               = CLOCK_SOURCE_IS_CONTINUOUS |
            CLOCK_SOURCE_MUST_VERIFY,
   .archdata            = { .vclock_mode = VCLOCK_TSC },
};

static DECLARE_DELAYED_WORK(tsc_irqwork,
     tsc_refine_calibration_work);
static int __init init_tsc_clocksource(void)
{
   if (!cpu_has_tsc || tsc_disabled > 0 || !tsc_khz)
      return 0;

   if (tsc_clocksource_reliable)
      clocksource_tsc.flags &= ~CLOCK_SOURCE_MUST_VERIFY;
   /* lower the rating if we already know its unstable: */
   if (check_tsc_unstable()) {
      clocksource_tsc.rating = 0;
      clocksource_tsc.flags &= ~CLOCK_SOURCE_IS_CONTINUOUS;
   }

   if (boot_cpu_has(X86_FEATURE_NONSTOP_TSC_S3))
      clocksource_tsc.flags |= CLOCK_SOURCE_SUSPEND_NONSTOP;

   /*
    * Trust the results of the earlier calibration on systems
    * exporting a reliable TSC.
    */
   if (boot_cpu_has(X86_FEATURE_TSC_RELIABLE)) {
      clocksource_register_khz(&clocksource_tsc, tsc_khz);
      return 0;
   }

   schedule_delayed_work(&tsc_irqwork, 0);
   return 0;
}
```

[20] clocksource.h on Bootlin Elixir
[21] clocksource.h on Bootlin Elixir
[22] clocksource.h on Bootlin Elixir
[23] jiffies.c on Bootlin Elixir
[24] hpet.c on Bootlin Elixir
[25] kvmclock.c on Bootlin Elixir
[26] tsc.c on Bootlin Elixir
[27] tsc.c on Bootlin Elixir

A.5 PROCESS_CPUTIME_ID: Scheduler

As mentioned above, the CLOCK_PROCESS_CPUTIME_ID getter invokes the method thread_group_cputime[28]:

```
/*
 * Accumulate raw cputime values of dead tasks
 * (sig->[us]time) and live tasks (sum on group iteration)
 * belonging to @tsk's group.
 */
void thread_group_cputime(struct task_struct *tsk, struct
     task_cputime *times)
{
   struct signal_struct *sig = tsk->signal;
   cputime_t utime, stime;
   struct task_struct *t;
   unsigned int seq, nextseq;
   unsigned long flags;

   rcu_read_lock();
   /* Attempt a lockless read on the first round. */
   nextseq = 0;
   do {
      seq = nextseq;
      flags =
           read_seqbegin_or_lock_irqsave(&sig->stats_lock,
           &seq);
      times->utime = sig->utime;
      times->stime = sig->stime;
      times->sum_exec_runtime = sig->sum_sched_runtime;

      for_each_thread(tsk, t) {
         task_cputime(t, &utime, &stime);
         times->utime += utime;
         times->stime += stime;
         times->sum_exec_runtime += task_sched_runtime(t);
      }
      /* If lockless access failed, take the lock. */
      nextseq = 1;
   } while (need_seqretry(&sig->stats_lock, seq));
   done_seqretry_irqrestore(&sig->stats_lock, seq, flags);
   rcu_read_unlock();
}
```

In turn, this computes sum_exec_runtime (relevant for our measurements) through calls to task_sched_runime(t)[29]. This quantity is managed by the Linux scheduler.

```
/*
 * Return accounted runtime for the task.
 * In case the task is currently running, return the
 * runtime plus current's pending runtime that have not
 * been accounted yet.
 */
unsigned long long task_sched_runtime(struct task_struct *p)
{
   unsigned long flags;
   struct rq *rq;
   u64 ns;

#if defined(CONFIG_64BIT) && defined(CONFIG_SMP)
   if (!p->on_cpu || !task_on_rq_queued(p))
      return p->se.sum_exec_runtime;
#endif

   rq = task_rq_lock(p, &flags);
   if (task_current(rq, p) && task_on_rq_queued(p)) {
      update_rq_clock(rq);
      p->sched_class->update_curr(rq);
   }
   ns = p->se.sum_exec_runtime;
   task_rq_unlock(rq, p, &flags);

   return ns;
}
```

The important (clock-related) step in the code above is the call to the function update_rq_clock[30] which calls sched_clock_cpu[31]. Finally, (assuming the scheduler clock is stable), this is a wrapper around the sched_clock function.

```
void update_rq_clock(struct rq *rq)
{
   s64 delta;
   lockdep_assert_held(&rq->lock);
   if (rq->clock_skip_update & RQCF_ACT_SKIP)
      return;
   delta = sched_clock_cpu(cpu_of(rq)) - rq->clock;
   if (delta < 0)
      return;
   rq->clock += delta;
   update_rq_clock_task(rq, delta);
}

u64 sched_clock_cpu(int cpu)
{
   struct sched_clock_data *scd;
   u64 clock;
   if (sched_clock_stable())
      return sched_clock();
   if (unlikely(!sched_clock_running))
      return 0ull;

   preempt_disable_notrace();
   scd = cpu_sdc(cpu);
   if (cpu != smp_processor_id())
      clock = sched_clock_remote(scd);
   else
      clock = sched_clock_local(scd);
   preempt_enable_notrace();

   return clock;
}
```

The situation is now very similar to the REALTIME/MONOTONIC clocks. The default sched_clock implementation[32] is based on Jiffies, and

[28] cputime.c on Bootlin Elixir
[29] core.c on Bootlin Elixir
[30] core.c on Bootlin Elixir
[31] clock.c on Bootlin Elixir
[32] clock.c on Bootlin Elixir

is expected to be overwritten later on by some more accurate time-source.

```
/*
 * Scheduler clock - returns current time in nanosec units.
 * This is default implementation.
 * Architectures and sub-architectures can override this.
 */
unsigned long long __weak sched_clock(void)
{
   return (unsigned long long)(jiffies - INITIAL_JIFFIES)
               * (NSEC_PER_SEC / HZ);
}
```

On x86, this ends up being the case when the TSC overwrites `sched_clock`[33]. So, unless something about the TSC is misconfigured or misbehaving, the scheduler clock and `CLOCK_PROCESS_CPUTIME_ID` will also be based on the TSC.

```
unsigned long long sched_clock(void)
{
   return paravirt_sched_clock();
}

struct pv_time_ops pv_time_ops = {
   .sched_clock = native_sched_clock,
   .steal_clock = native_steal_clock,
};
static inline unsigned long long paravirt_sched_clock(void)
{
   return PVOP_CALL0(unsigned long long,
        pv_time_ops.sched_clock);
}

/*
 * Scheduler clock - returns current time in nanosec units.
 */
u64 native_sched_clock(void)
{
   if (static_branch_likely(&__use_tsc)) {
      u64 tsc_now = rdtsc();

      /* return the value in ns */
      return cycles_2_ns(tsc_now);
   }
   /*
    * Fall back to jiffies if there's no TSC available:
    * ( But note that we still use it if the TSC is marked
    *   unstable. We do this because unlike Time Of Day,
    *   the scheduler clock tolerates small errors and it's
    *   very important for it to be as fast as the platform
    *   can achieve it. )
    */
   /* No locking but a rare wrong value is not a big deal: */
   return (jiffies_64 - INITIAL_JIFFIES) * (1000000000 / HZ);
}
```

Received 5 May 2024; revised 5 May 2024; accepted 5 May 2024

[33] tsc.c, paravirt.h, paravirt.c and tsc.c on Bootlin Elixir