

Goldilocks and the Three P-States: Mitigating Hertzbleed with Formal Leakage Guarantees

Inwhan Chun*
Carnegie Mellon University
inwhanc@andrew.cmu.edu

Christine Guo*
Princeton University†
cg4302@princeton.edu

Riccardo Paccagnella
Carnegie Mellon University
rpaccagn@cs.cmu.edu

Abstract—Hertzbleed is an emerging class of remote timing attacks that can leak secrets previously considered beyond the reach of timing analysis. The attack exploits how, when a processor exceeds power or thermal limits and starts throttling, CPU frequency—and thus, program runtime—becomes dependent on power consumption. In response to Hertzbleed, several software-level mitigations have been proposed, including masking, key refresh, noise injection, and disabling frequency boost. However, none of these mitigations achieves general software applicability, low overhead, and provable security.

In this work, we introduce Goldilocks, a practical mitigation against Hertzbleed. Goldilocks treats Hertzbleed as an information-theoretic channel and limits how much information the channel can carry by constraining when and how throttling can occur. It can be deployed on existing processors with no changes to application software, maintains a CPU frequency level that is “just right” for each machine and workload, and provides formal leakage bounds that reduce worst-case leakage growth from linear in execution time to as little as logarithmic. Our evaluation across a variety of processors and workloads shows that Goldilocks effectively mitigates Hertzbleed attacks and incurs low overhead.

1. Introduction

Hertzbleed is a recently uncovered class of attacks that turns power side-channel attacks into remote timing attacks [1]–[5]. The attack exploits how, when modern processors exceed power or thermal limits and start throttling, their dynamic frequency scaling feature becomes dependent on power consumption. This directly translates to data-dependent execution time differences, which are remotely observable. Hertzbleed undermines established mitigations against both power side channels and timing attacks. For example, it enables remote key extraction attacks even from constant-time cryptographic code [1]–[3], [5], [6].

In response to Hertzbleed, several software-level mitigations have been proposed. In security advisories, hardware vendors have recommended software-based masking, frequent key refreshes, and random noise injection, all of which aim to reduce the correlation between secret data

and power consumption (and hence CPU frequency) during execution [7]–[11]. In research papers, blog posts, and mailing lists, researchers have advocated lowering the highest allowed CPU frequency (e.g., to the base frequency) to reduce the likelihood of frequency throttling, which is the root cause of Hertzbleed [1], [4], [5], [12]–[14].

Yet, these mitigations have limited practicality. Masking is algorithm-specific, incurs high overhead (e.g., due to its reliance on fresh randomness), and may still leak in practice [15]–[23]. Frequent key refreshes are not always viable, especially in applications that rely on long-term secrets [8], [14], [24]. Random noise can be averaged out by collecting additional samples, since the baseline power consumption remains secret dependent [25]–[28]. Finally, fixing the highest allowed CPU frequency in a way that both prevents throttling and preserves performance is nontrivial, since the level that achieves this balance varies across machines and workloads, forcing a trade-off between performance and security: setting it too high may not prevent leakage, while setting it too low may significantly degrade performance.¹ As a result, no existing mitigation achieves general software applicability, low overhead, and provable security.

In this work, we introduce *Goldilocks*, a practical mitigation against Hertzbleed. Goldilocks treats Hertzbleed as an information-theoretic channel and limits how much information the channel can carry by constraining when and how throttling can occur. The approach can be deployed on existing processors with no changes to application software. It incurs low overhead by maintaining a CPU frequency level that is “just right” for each machine and workload—neither too low (degrading performance) nor too high (causing throttling). And it provides provable security by limiting how much information the Hertzbleed side channel can leak over time, reducing worst-case leakage growth from linear in execution time to as little as logarithmic.

Inspired by prior work on modeling side channels [29]–[34], we first model Hertzbleed as an information-theoretic channel from secrets to P-state (i.e., frequency) sequences. In this view, leakage is governed by how many distinct P-state sequences can arise during an execution. We prove that

1. Fully preventing throttling may require fixing the CPU frequency to its minimum allowed value, as done by some [14]. This is because, as we show in Section 8, data-dependent throttling may occur even when the highest allowed CPU frequency is set to the base frequency.

*These authors contributed equally to this work.

†Work partially done while at Carnegie Mellon University.

in the unmitigated case, throttling can induce many such sequences, yielding worst-case leakage that grows linearly with execution time t (i.e., $O(t)$). However, if one could constrain the total amount of throttling during an execution, worst-case leakage would grow only logarithmically with t .

To enforce such constraints in practice, Goldilocks divides time into *epochs*, each governed by a *P-state schedule* chosen at the start of the epoch. While the processor does not throttle, an epoch follows its P-state schedule. When throttling occurs, Goldilocks ends the current epoch, enforces a cooldown period to stop throttling, and starts a new epoch with an updated (typically more conservative) P-state schedule. Over time, this adaptive control loop lets Goldilocks converge to a P-state schedule that is “just right” for the machine and workload. At the same time, by limiting how many epochs can occur and how long throttling can last within each epoch, Goldilocks constrains the number of realizable P-state sequences (and thus leakage).

We show that Goldilocks yields strong, configurable bounds on Hertzbleed leakage. Specifically, with Goldilocks, worst-case leakage grows as $O(N \log t)$ (as opposed to $O(t)$ in the unmitigated case), where N is the maximum number of epochs that could have occurred up to t . When Goldilocks is configured to never reuse P-states that have been observed to cause throttling, N is a constant, and leakage grows only logarithmically with t . When Goldilocks is configured to periodically revisit P-states previously observed to cause throttling (to recover performance), N can grow with t , but the defender controls how fast it grows by choosing how often revisits occur. Our analysis also provides a way to enforce arbitrary bounds on leakage: given machine and configuration parameters, Goldilocks can conservatively track worst-case leakage during execution and switch to a fallback safe mode once the target leakage budget is reached.

We implement a prototype of Goldilocks as a Linux kernel module for Intel and AMD processors and evaluate it across a range of client and server processors and cryptographic workloads. The prototype monitors throttling and enforces P-state schedules using existing processor interfaces and includes several simple strategies for converging to a “just right” P-state schedule, along with optional mechanisms to revisit higher P-states over time. Our experiments demonstrate that, once converged, Goldilocks incurs low overhead relative to an unmitigated baseline, while outperforming naive frequency-capping approaches (such as fixing the CPU at its base or minimum frequency). Moreover, Goldilocks can, in some cases, even improve energy efficiency relative to an unmitigated baseline. When cryptographic workloads run alongside other workloads or power headroom changes over time, we show that overhead can increase temporarily, but the revisit mechanisms help recover performance. Finally, we reproduce published proof-of-concept Hertzbleed attacks against SIKE, ECDSA, and Classic McEliece and show that Goldilocks eliminates the secret-dependent frequency signal that they exploit.

2. Background

2.1. Processor power management

Modern processors control CPU frequency at the granularity of *P-states*, where each P-state corresponds to a discrete frequency level. For example, modern Intel CPUs adjust frequency in 100 MHz steps [35], while AMD CPUs use 25 MHz steps [36]. Like prior work [1], [3]–[5], we adopt Linux’s P-state naming convention, where higher P-states correspond to higher frequencies [37]. Each CPU has a *base frequency* (called the nominal frequency on AMD CPUs [38]), which is the frequency it should be able to sustain under normal operating conditions without exceeding its Thermal Design Power (TDP) [39]. The Linux `CPUFreq` subsystem enables restricting the range of usable P-states (e.g., by reducing the highest allowed P-state) [40].

To stay within safe power and thermal limits, modern processors dynamically adjust their P-states at runtime. Internally, a power management algorithm periodically evaluates whether any *reactive limits* (e.g., power or temperature thresholds) are exceeded based on running averages of electrical and thermal parameters [2]. We refer to the fixed time interval at which this control loop can update the P-state as a *time quantum*. When no reactive limit is exceeded, the CPU may run at the highest P-state permitted by the operating system. When a limit is reached, the processor starts *throttling*: the hardware reactively reduces the P-state so that operation remains within predefined safe operating conditions. Many modern processors can deliver an interrupt to the operating system when throttling starts.

2.2. Hertzbleed attacks

As discussed above, once a processor exceeds reactive limits and starts throttling, the hardware lowers the CPU P-state to stay within a safe power and thermal budget. This budget depends on the processor’s recent power consumption, making the P-states during throttling a function of that power consumption [1], [2]. Thus, if a victim’s computation consumes different amounts of power for different secret values, those secrets can result in different CPU P-states during the victim’s execution. An adversary who can observe these P-states—either directly (by monitoring frequency) or indirectly (via execution time, since runtime depends on frequency)—can therefore learn information about the secret. Prior work on *Hertzbleed* has shown that this data-dependent throttling can undermine established mitigations against both power side channels and timing attacks [1]–[3]. For example, it can be used to leak keys from constant-time cryptographic code [1]–[3], [5], revive pixel stealing attacks [3]–[5], fingerprint websites [4], break KASLR [1], and mount Meltdown- and MDS-style attacks [41].

2.3. Entropy and information theory

Shannon entropy quantifies the uncertainty associated with the outcome of a random variable. Suppose X is a

discrete random variable that takes values in \mathcal{X} , where each outcome $x \in \mathcal{X}$ occurs with probability $p(x)$. The Shannon entropy of X , representing the number of possible bits of information encoded by the outcomes of X , is:²

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \log p(x).$$

The entropy of X is bounded by:

$$H(X) \leq \log(|\mathcal{X}|), \quad (1)$$

where $|\mathcal{X}|$ is the number of distinct outcomes of X . This shows that while the information capacity of a channel depends on the distribution of X over \mathcal{X} , its maximum is achieved when X is uniformly distributed. Equation 1 is widely used to quantify worst-case information leakage via side channels, as it provides an upper bound on how many bits an attacker could encode in or extract through the channel in the worst case [29]–[34].

Information-theoretic bits vs. secret bits. In this paper, we measure leakage in information-theoretic “bits,” i.e., using the entropy of an observable distribution. These bits quantify statistical information: they upper bound how much information an adversary *could* encode in and transmit through the channel in the worst case. They do not directly correspond to the number of secret bits an attacker can necessarily recover in a concrete attack, since practical recovery also depends on how secrets are encoded into the channel and on measurement noise. Consequently, our entropy-based formulas should be interpreted as conservative (worst-case) upper bounds on channel capacity.

3. Assumptions and threat model

We consider an *adversary* whose goal is to recover secret data processed by a *victim* program. We assume that the victim’s computations on secret data affect the processor’s power consumption and that, due to Hertzbleed, power consumption in turn affects the CPU’s P-states. We assume that the CPU’s P-states depend on secret data only through power consumption³ and that the adversary can observe all P-states the CPU runs at during the victim’s execution. We make no further assumptions about the adversary’s strategy or the nature of the victim’s computations. This threat model also captures a weaker remote-timing scenario, where the adversary observes CPU frequency variations indirectly through timing,⁴ and a strong covert-channel scenario, where the victim (sender) and the adversary (receiver) collude to maximize the amount of information encoded into the channel.

2. All logarithms in this paper are base 2.

3. On some processors, P-state selection may also depend on factors such as the number of active cores [35], [42]. These factors are orthogonal to Hertzbleed’s power-to-frequency mechanism, so we condition on them and analyze leakage only from Hertzbleed-induced P-state variations.

4. In Section 9, we discuss how timing channels instantiate this model.

4. Motivation: Quantifying Hertzbleed leakage

In this section, we model Hertzbleed as an information-theoretic channel and quantify how much information it can carry in the absence of a mitigation. This model serves as the baseline for the analysis of Goldilocks in Section 6.

Notation. Let P be the finite set of P-states supported by the CPU and $|P|$ denote its size. The P-states the CPU runs at during the victim’s execution can be represented as a sequence of P-states (p_1, p_2, \dots, p_t) , where $p_i \in P$ for each i is the P-state that the CPU runs at during time quantum i .⁵

Leakage analysis. As described in Section 2.3, to quantify worst-case information leakage we must determine the number of distinct sequences (p_1, \dots, p_t) that are realizable during the victim’s execution. For the counting argument below, we make one simplifying assumption: the P-state behavior while the processor does not throttle is fixed.⁶ The number of realizable sequences therefore depends on how often throttling occurs during the victim’s execution. If throttling never occurs, only one sequence is possible. If throttling occurs during i out of the t time quanta, the number of realizable sequences is $\binom{t}{i}(|P| - 1)^i$, because there are $\binom{t}{i}$ ways to choose which time quanta throttle and each throttling quantum may take any of $|P| - 1$ throttling P-states. To model the worst case, we allow each time quantum to throttle independently—that is, each time quantum may independently take any P-state. Thus, the number of time quanta during which the processor throttles can be anywhere between 0 (no throttling) and t (always throttling), and the number of realizable P-state sequences of length t is:

$$\sum_{i=0}^t \binom{t}{i} (|P| - 1)^i = (1 + (|P| - 1))^t = |P|^t,$$

where the first equality follows directly from the binomial theorem. By Equation 1, the maximum leakage L is the logarithm of the number of distinct realizable sequences:

$$L = \log(|P|^t) = t \log |P|. \quad (2)$$

This shows that, in the absence of any mitigation, the maximum information that can be extracted through a P-state sequence grows *linearly* with execution time t .

Reducing leakage by limiting throttling. The $O(t)$ linear growth above arises from two degrees of freedom: the number of distinct P-states a (throttling) time quantum can take and the number of time quanta during which the processor can throttle. We now analyze the effect of limiting the latter through a hypothetical constraint that restricts throttling to at most $k \ll t$ time quanta during the victim’s execution. With this constraint, the number of realizable sequences is:

$$\sum_{i=0}^k \binom{t}{i} (|P| - 1)^i.$$

5. Without loss of generality, the time quanta are positive integers.

6. This does not affect the leakage counted here, because in our model no secret-dependent variation occurs before throttling begins.

Using the fact that $1 \leq \binom{t}{i} \leq t^i \leq t^k$ for $i \leq k^7$ and the properties of finite geometric series with first term 1 and common ratio $|P| - 1$, we can write (assuming $|P| \geq 3$):

$$\begin{aligned} \sum_{i=0}^k \binom{t}{i} (|P| - 1)^i &\geq \sum_{i=0}^k (|P| - 1)^i \geq (|P| - 1)^k; \\ \sum_{i=0}^k \binom{t}{i} (|P| - 1)^i &\leq \sum_{i=0}^k t^k (|P| - 1)^i = t^k \sum_{i=0}^k (|P| - 1)^i \\ &= t^k \frac{(|P| - 1)^{k+1} - 1}{(|P| - 1) - 1} \leq t^k (|P| - 1)^{k+1}. \end{aligned}$$

Taking logarithms (as per Equation 1) yields:

$$k \log(|P| - 1) \leq L \leq (k + 1) \log(|P| - 1) + k \log t.$$

This shows that, if one can limit the number of throttling time quanta to at most k , leakage grows linearly in k and only at most logarithmically in t . The takeaway is that constraining how often throttling can occur reduces the effective channel capacity, slowing the growth of information leakage.

We emphasize that the above constraint is just an illustrative abstraction. Later, in Section 6, we analyze the concrete mechanisms by which Goldilocks enforces constraints of a similar nature (on how many times throttling can occur and on how long throttling can last), and we use a refined model to quantify the resulting asymptotic leakage.

5. Design of Goldilocks

In this section, we introduce Goldilocks, a practical mitigation against Hertzbleed. At a high level, Goldilocks’ security goal is to constrain how often and for how long frequency throttling can occur, thereby reducing the number of realizable P-state sequences and the corresponding leakage.

5.1. Design goals

More concretely, our goal is to design a mitigation against Hertzbleed that satisfies the following properties:

- G1 Provable security.** The mitigation must be able to measure and provably limit the worst-case information leakage from a P-state sequence. Moreover, it must allow users to configure a target leakage budget and enforce that the resulting sequence does not exceed it.
- G2 General applicability.** The mitigation must protect arbitrary applications without requiring application-level changes, and it must be deployable on existing off-the-shelf processors without hardware modifications.
- G3 Low overheads.** The mitigation must incur low overhead: for reasonable leakage budgets, it should not substantially degrade performance or increase energy consumption compared to an unmitigated system.

As discussed in Section 1, no existing Hertzbleed mitigation satisfies these three properties simultaneously.

7. $\binom{t}{i} = \frac{t(t-1)\cdots(t-i+1)}{i!} \leq t(t-1)\cdots(t-i+1) \leq t^i$.

Software-based masking is algorithm-specific and incurs high overhead, violating G2 and G3. Frequent key refreshes are not viable in all applications (e.g., in applications that rely on long-term secrets), violating G2. Random noise can be averaged out by collecting additional samples, violating G1. Finally, fixing the highest allowed CPU frequency in a way that both prevents throttling and preserves performance is nontrivial, since the level that achieves this balance varies across machines and workloads, forcing a trade-off between performance and security: setting it too high may not prevent leakage (violating G1), while setting it too low (e.g., to the minimum frequency, as discussed in Footnote 1) may significantly degrade performance (violating G3).

5.2. Goldilocks mechanism

Goldilocks resolves the trade-off between performance and security faced by prior Hertzbleed mitigations by finding a P-state value that is “just right” for each machine and workload—neither too low (degrading performance) nor too high (exceeding reactive limits and causing throttling).

Its core mechanism is an adaptive frequency control loop that divides time into *epochs*, each governed by a *P-state schedule*. When frequency throttling occurs, Goldilocks detects it, terminates the current epoch, and starts a new one with an updated (typically more conservative) P-state schedule. Epochs enable Goldilocks to find the highest P-state a workload can sustain without throttling and back off whenever throttling reveals that the current setting is too aggressive. By repeatedly adjusting P-state schedules in response to frequency throttling, Goldilocks can converge to a P-state schedule that is “just right” for the machine and workload, achieving G3. At the same time, by limiting how many epochs can occur and how long frequency throttling can last within each epoch, Goldilocks constrains the number of realizable P-state sequences, achieving G1.

We define an epoch as a contiguous interval of time quanta governed by a P-state schedule chosen by Goldilocks at the beginning of the epoch. This schedule may be as simple as a constant value or vary over time (e.g., gradually increasing or decreasing the P-state), as long as the changes are predetermined.⁸ Within an epoch, while the processor does not throttle, the P-states follow the chosen schedule exactly. If the processor exceeds its reactive limits, however, the hardware may override the schedule and start throttling. When this occurs, Goldilocks initiates an epoch transition.

Figure 1 illustrates an example of an epoch transition. Suppose Goldilocks starts an epoch with a constant P-state schedule set to the maximum supported P-state (①). The processor follows this schedule for some time. Eventually, the processor starts throttling (②). While the processor throttles, the hardware may select lower P-states for a brief period. Goldilocks detects this throttling, enforces a fixed cooldown period (③), and starts a new epoch with an updated, more conservative P-state schedule (④).

8. For example, a schedule that varies the P-state over time based on the processor temperature would be invalid, because it is not predetermined.

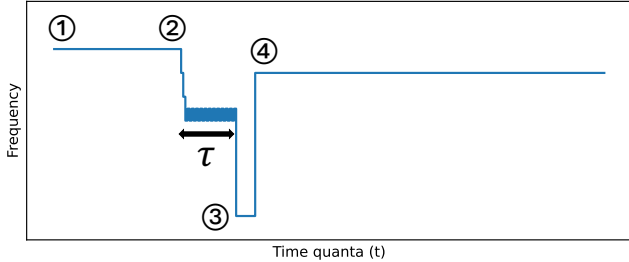


Figure 1. Example of an epoch transition. The system follows a predetermined P-state schedule until throttling is detected (①). The hardware may then select lower P-states for up to τ time quanta (②), after which Goldilocks enforces a fixed cooldown period (③) and starts a new epoch with an updated (more conservative) P-state schedule (④).

In practice (as seen in Figure 1), Goldilocks may not be able to react to throttling instantaneously: there may be a delay between the moment throttling starts, the moment Goldilocks detects it, and the moment the new P-state schedule is enforced. We denote by τ a conservative upper bound on the number of time quanta during which the hardware may select throttling P-states after throttling starts and before Goldilocks reacts. τ is a processor-specific constant that we determine empirically in Section 7.

Moreover, to avoid starting new epochs while the processor is still above its reactive limits, Goldilocks inserts a *cooldown period* between epochs. During cooldown, Goldilocks enforces a predetermined low-power P-state that allows the processor to return within safe limits before the next epoch begins: for example, Goldilocks may force the P-state to the minimum supported P-state for a fixed number of time quanta. Because the cooldown behavior is deterministic, it does not increase the number of realizable P-state sequences and does not affect leakage.

The Goldilocks mechanism described above serves two purposes. First, it constrains information leakage: as we prove in Section 6, limiting the number of epochs and the throttling duration per epoch reduces worst-case leakage growth from $O(t)$ in the unmitigated case to $O(N \log t)$ under Goldilocks, where t is the execution time and N is the maximum number of epochs that could have occurred so far. Second, it enables a simple mechanism to track and bound leakage online, since leakage depends only on four quantities: the current execution time t , the maximum number of epochs N , the number of P-states $|P|$, and the maximum number of throttling time quanta per epoch τ .

5.3. Goldilocks policy

A Goldilocks *policy* specifies how to choose the P-state schedule for each epoch and what happens when a leakage budget is reached. Policies can be static or adaptive.

Static policies. In a *static* policy, Goldilocks never reuses a P-state that has been observed to cause throttling: whenever throttling is observed at some P-state k , all subsequent epochs are restricted to use only P-states strictly below k in

their P-state schedules. In other words, the policy maintains a strictly decreasing *upper bound* on which P-states may appear in future P-state schedules, and each epoch transition eliminates at least one P-state from future consideration. For example, a policy where each epoch uses a single P-state in its schedule may decrease that P-state by one at each epoch transition (as seen in Figure 1). As we show in Section 6, static policies yield worst-case leakage that grows at most logarithmically with execution time.

Adaptive policies. In an *adaptive* policy, Goldilocks may revisit P-states that previously caused throttling. Specifically, when selecting the P-state schedule for a new epoch, it may choose a schedule that—after a policy-defined interval without throttling—steps to a P-state above the current upper bound (i.e., the highest P-state that has not been observed to cause throttling). This can improve performance if a P-state that previously caused throttling no longer does so under the current conditions. However, if throttling occurs again, it triggers an additional epoch transition after which Goldilocks must revert to the previous upper bound.

Because throttling can occur more than once per P-state, adaptive policies can introduce additional epoch transitions (and thus leakage) compared to static policies. Nevertheless, the policy controls when P-state schedules include revisits, allowing users to tune the trade-off between performance and leakage. For example, a P-state schedule may include a revisit after each hour without throttling or at exponentially increasing times (e.g., after 1, 2, 4, 8, . . . hours without throttling). As we discuss in Section 6, different revisit patterns trade off performance recovery against leakage.

In addition to controlling *when* a P-state schedule includes revisits, an adaptive policy also controls *which* higher P-state to try (e.g., stepping up by one or restoring an earlier upper bound). A policy may condition these choices on non-secret information available before the epoch begins (e.g., which types of workloads are co-scheduled with the victim). Importantly, the choice of which higher P-state to try does not change leakage accounting, which upper-bounds leakage by conservatively assuming every revisit causes throttling.

Leakage budget. Policies also specify what happens when a leakage budget is reached. For example, a policy can instruct Goldilocks to switch to a fallback safe mode (e.g., setting the P-state to the minimum supported one) once the online leakage calculation exceeds a configured threshold. We describe how such budgets are evaluated in Section 6.

We emphasize that Goldilocks’ leakage tracking mechanism is agnostic to the policy. Regardless of how P-state schedules are chosen, leakage depends solely on t , N , $|P|$, and τ . This makes it easy for Goldilocks to track leakage for arbitrary policies and to enforce target leakage budgets.

5.4. System requirements

We now discuss what Goldilocks requires from the underlying hardware and operating system (OS), and why these requirements are satisfied by commodity platforms.

At a high level, Goldilocks needs two capabilities: (i) a way to detect when the processor throttles, and (ii) a way to enforce the P-state choices dictated by the current schedule. These capabilities are already available in modern platforms.

On the detection side, many modern processors can deliver an interrupt to the OS when throttling occurs. On platforms without such an interrupt, throttling can be inferred by monitoring the P-state over time and detecting drops relative to the configured maximum. Goldilocks can use either mechanism as the trigger for epoch transitions.

On the control side, modern processors support interfaces to constrain the P-states that may be used during execution. On Linux, these interfaces are exposed via the `CPUFreq` subsystem. Goldilocks uses them to (i) enforce a fixed cooldown period between epochs and (ii) realize the P-state schedules dictated by the policy.

As a result, Goldilocks can be implemented purely in software, e.g., as a kernel module. No changes to application code or hardware are required, satisfying G2.

6. Security analysis

In this section, we derive an upper bound on the information-theoretic leakage that can be extracted through Hertzbleed in the presence of Goldilocks. Our analysis demonstrates that Goldilocks significantly reduces information leakage compared to the unmitigated case (Section 4).

Notation. We use the same notation as Section 4. Additionally, we let τ denote the maximum number of time quanta during which the processor may continue to select throttling P-states after throttling starts and before Goldilocks reacts, as introduced in Section 5.2. We let N denote the maximum number of epochs that could have occurred up to time t ; for static policies, N is a constant, while for adaptive policies N may itself be a function of t . We emphasize that our analysis holds regardless of the values of τ and $|P|$, which are machine specific, and N , which depends on the policy.

Leakage analysis. Like in Section 4, to quantify worst-case information leakage we must determine the number of distinct P-state sequences that are realizable during the victim's execution. In the presence of Goldilocks, it is convenient to abstract away the deterministic cooldown segments (which always enforce the same P-state and do not increase the number of realizable sequences) and focus only on the remaining parts of the sequence. The resulting sequence can be modeled as a concatenation of n subsequences of lengths $t_1, \dots, t_n \in \mathbb{Z}^+$ satisfying $\sum_{k=1}^n t_k = t$, where each subsequence corresponds to a single epoch:

$$\begin{aligned} (p_1, \dots, p_t) &= (p_1, \dots, p_{t_1}) \\ &\quad || (p_{t_1+1}, \dots, p_{t_1+t_2}) \\ &\quad || \dots || (p_{\sum_{k=1}^{n-1} t_k+1}, \dots, p_{\sum_{k=1}^n t_k}). \end{aligned}$$

To determine the number of distinct realizable sequences (and thus worst-case leakage), we need to quantify the number of possible values and lengths for each epoch.

Recall that in each epoch, Goldilocks follows a deterministic P-state schedule until the processor starts throttling, after which it can take any of the throttling P-state values for the remaining time quanta. Let i be the number of time quanta during which the processor throttles in a given epoch k of length t_k . With Goldilocks, i can be anywhere between 0 and $\min(t_k - 1, \tau)$. Each throttling time quantum may take at most $|P| - 1$ values, and thus the number of possible values such an epoch can take is bounded by:

$$\sum_{i=0}^{\tau} (|P| - 1)^i.$$

If we let $\mathcal{T}[t, n] = \{(t_1, \dots, t_n) \in \mathbb{Z}^+{}^n \mid \sum_{k=1}^n t_k = t\}$ denote the set of all possible epoch lengths for a sequence composed of n epochs, the number of distinct realizable sequences composed of at most N epochs is bounded by:

$$\sum_{n=1}^N \sum_{(t_1, \dots, t_n) \in \mathcal{T}[t, n]} \prod_{k=1}^n \left(\sum_{i=0}^{\tau} (|P| - 1)^i \right), \quad (3)$$

where the outer sum is over the possible number of epochs n , the inner sum is over the ways to partition t into n epoch lengths t_1, \dots, t_n , and the product gives the number of possible values for a sequence composed of those n epochs.

To reason about the asymptotic growth of Equation 3 more easily, we now compute an upper bound on it. First, the properties of finite geometric series with first term 1 and common ratio $|P| - 1$ yield (assuming $|P| \geq 3$):

$$\sum_{i=0}^{\tau} (|P| - 1)^i = \frac{(|P| - 1)^{\tau+1} - 1}{(|P| - 1) - 1} \leq (|P| - 1)^{\tau+1}.$$

Second, using the above inequality, and since $\sum_{k=1}^n t_k = t$ and, by combinatorics, $|\mathcal{T}[t, n]| = \binom{t-1}{n-1}$,⁹ we can write:

$$\begin{aligned} &\sum_{n=1}^N \sum_{(t_1, \dots, t_n) \in \mathcal{T}[t, n]} \prod_{k=1}^n \left(\sum_{i=0}^{\tau} (|P| - 1)^i \right) \\ &\leq \sum_{n=1}^N \sum_{(t_1, \dots, t_n) \in \mathcal{T}[t, n]} \prod_{k=1}^n (|P| - 1)^{\tau+1} \\ &= \sum_{n=1}^N \sum_{(t_1, \dots, t_n) \in \mathcal{T}[t, n]} (|P| - 1)^{(\tau+1)n} \\ &= \sum_{n=1}^N \binom{t-1}{n-1} (|P| - 1)^{(\tau+1)n}. \end{aligned}$$

9. This can be derived using the stars and bars theorem.

Third, again using the properties of finite geometric series with first term and common ratio $(|P|-1)^{\tau+1}$, we can write:

$$\begin{aligned} \sum_{n=1}^N (|P|-1)^{(\tau+1)n} &= \frac{(|P|-1)^{\tau+1}((|P|-1)^{(\tau+1)N} - 1)}{(|P|-1)^{\tau+1} - 1} \\ &\leq \frac{(|P|-1)^{\tau+1}(|P|-1)^{(\tau+1)N}}{(|P|-1)^{\tau+1} - 1} \\ &= \frac{(|P|-1)^{(\tau+1)(N+1)}}{(|P|-1)^{\tau+1} - 1} \\ &\leq (|P|-1)^{(\tau+1)(N+1)}. \end{aligned}$$

Using the above three equations and the fact that $\binom{t-1}{n-1} \leq t^{n-1} \leq t^{N-1}$ for $n \leq N$, it therefore holds that the total number of distinct realizable sequences composed of at most N epochs (Equation 3) is bounded by:

$$\begin{aligned} &\sum_{n=1}^N \binom{t-1}{n-1} (|P|-1)^{(\tau+1)n} \\ &\leq \sum_{n=1}^N t^{N-1} (|P|-1)^{(\tau+1)n} \\ &= t^{N-1} \sum_{n=1}^N (|P|-1)^{(\tau+1)n} \\ &\leq t^{N-1} (|P|-1)^{(\tau+1)(N+1)}. \end{aligned} \quad (4)$$

We can now proceed with the calculation of the worst-case leakage. By Equation 1, the maximum leakage L is the logarithm of the number of distinct realizable sequences. Taking the logarithm of Equation 4 yields:

$$L \leq (N-1) \log t + (\tau+1)(N+1) \log(|P|-1). \quad (5)$$

This shows that, with Goldilocks, the information that can be extracted through a P-state sequence grows at most linearly with the maximum number of epochs N and logarithmically with execution time t , with $|P|$ and τ as machine-specific constants. Hence, Goldilocks reduces worst-case information leakage growth from $O(t)$ (unmitigated) to $O(N \log t)$.

Static vs adaptive policies. The $O(N \log t)$ bound above shows that leakage grows linearly with the maximum number of epochs N , which in turn is determined by the Goldilocks policy (Section 5.3). In a static policy, once Goldilocks observes throttling at a P-state k , all subsequent epochs are restricted to use only P-states strictly below k . Because this upper bound decreases monotonically across epochs, each epoch transition eliminates at least one P-state from future consideration. Therefore, $N \leq |P|$ and, since $|P|$ is a constant, worst-case leakage grows at most logarithmically with execution time t , as visualized in Figure 2.

In adaptive policies, Goldilocks may occasionally revisit higher P-states. In this case, N can grow with t . How fast N grows depends on how often P-state schedules include revisits: in the worst case, each revisit causes throttling and

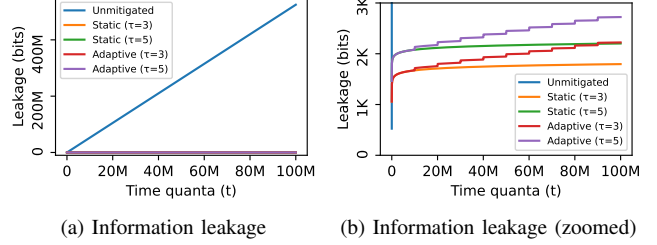


Figure 2. Information leakage bounds versus execution time t computed using Equations 2 (without Goldilocks) and 5 (with Goldilocks). We set $|P| = 38$ (the number of P-states supported by our i9-9900 processor) and consider both $\tau = 3$ (the value we observe on our i9-9900) and $\tau = 5$ (the largest value we observe across our processors; see Section 8). For the static policy, we assume that each epoch transition eliminates exactly one P-state from future consideration (thus, $N = |P|$). For the adaptive policy, we additionally assume that the P-state schedules include revisits each 10^7 time quanta without throttling; since each revisit can (in the worst case) induce one additional epoch transition, we conservatively use $N = |P| + \lceil t/10^7 \rceil$. The second plot is a zoomed-in version of the first.

adds one additional epoch transition.¹⁰ For example, given a policy in which P-state schedules include revisits each 10^7 time quanta (roughly 3 h assuming 1 ms per quantum) without throttling, it holds that $N = |P| + \lceil t/10^7 \rceil$: there are at most $|P|$ epoch transitions that monotonically lower the upper bound, plus at most $\lceil t/10^7 \rceil$ transitions caused by revisits. The resulting leakage still grows significantly slower than the unmitigated $O(t)$ growth in practice (Figure 2). Alternatively, consider a policy in which the interval without throttling required before a revisit doubles after each revisit occurs. For example, the policy may allow a revisit after 1 hour without throttling, then after 2 additional hours without throttling, then after 4, 8, and so on. Then $N = O(|P| + \log t)$, yielding a tighter $O(\log^2 t)$ leakage bound. Other revisit patterns are also possible.

Budget enforcement. Independently of the chosen policy, Goldilocks can enforce a user-specified leakage budget. Given a target budget L^* (in bits), Goldilocks uses Equation 5 to conservatively estimate the worst-case leakage L online as a function of t , N , τ , and $|P|$. If L reaches L^* , Goldilocks enters a fallback safe mode that prevents further throttling by, for example, fixing the P-state to its minimum allowed value for the remainder of the victim’s execution. Thus, leakage growth saturates once the budget is reached.

7. Implementation

We implement a prototype of Goldilocks as a Linux kernel module. Once loaded, Goldilocks enforces the P-state schedule of each epoch as specified by the selected policy.

Throttling detection. The Intel processors in our experimental setup expose a hardware interrupt to notify the operating system when throttling starts [35], [43]. We hook

10. Importantly, N does not depend on which higher P-state is chosen at each revisit: while the choice of P-state affects whether throttling actually occurs, our analysis conservatively assumes that it always does.

into the corresponding interrupt handler so that Goldilocks can detect when the processor has started throttling.

The AMD processors in our setup do not expose an analogous interrupt.¹¹ For these processors, Goldilocks detects throttling heuristically by monitoring the CPU frequency. We poll the frequency once every millisecond by reading the `scaling_cur_freq` interface. When the CPU frequency stays below the maximum allowed frequency for five consecutive samples while the CPU is active, we consider the processor to be throttling. On AMD processors, we additionally set the governor to `performance` so that the active CPU cores always attempt to run at the highest allowed P-state when not throttling, making it easier to distinguish throttling from power-unrelated frequency scaling behavior.

In practice, we observe negligible differences between polling and interrupt-driven throttling detection. To evaluate this, we fix the CPU frequency to a value that does not throttle, measure the runtime of a fixed-size workload (e.g., SIKE), and compare the results when Goldilocks is loaded (monitoring for any throttling) versus when it is not. Across both platforms, Goldilocks incurs less than 0.1% runtime overhead regardless of the detection mechanism.

Epoch transition. When throttling is detected, Goldilocks performs an epoch transition as illustrated in Figure 1: it terminates the current epoch, enforces a cooldown phase, and then starts a new epoch under a new P-state schedule. We implement cooldown by temporarily forcing all CPU cores to run at their minimum supported frequency f_{min} (using the `scaling_max_freq` interface) for 4 seconds, which we empirically found to ensure the processor operates within safe power and thermal limits at the start of a new epoch. After cooldown, Goldilocks adjusts the P-state limits as needed to enforce the schedule chosen for the next epoch.

Static policies. Our prototype implements three simple static policies to search for the “just right” frequency, which we denote by p^* . Across all these policies, once a P-state has been observed to cause throttling in some epoch, future epochs never use that P-state or any higher one (Section 5.3). All policies start at the highest supported P-state and differ only in how they move through the remaining P-states:

- **Linear.** Each epoch runs at a single P-state. Upon throttling, the next epoch runs at the next lower P-state.
- **Doubling.** On each epoch transition, this policy decreases the P-state by a variable step size. The step size starts at 1 and doubles at each epoch transition (1, 2, 4, ...), allowing the policy to skip multiple P-states quickly. After a fixed interval without throttling (45 seconds in our prototype), the policy increases the P-state by one step at a time, but never above the lowest P-state that has been observed to cause throttling.
- **Binary.** This policy keeps track of the highest P-state known to be safe (i.e., the highest P-state that has not

caused throttling) and the lowest P-state known to cause throttling and chooses the next candidate P-state as the midpoint between them. After a fixed interval without throttling (45 seconds), it marks the current P-state as safe and moves to the next midpoint (if any).

Adaptive policies. Our prototype also implements a few optional adaptive policies. Recall from Section 5.3 that these policies are configured using two parameters: (1) *when* revisits occur and (2) *which* higher P-state is chosen at a revisit. For when revisits occur, our prototype includes revisits in P-state schedules after fixed intervals of 30 minutes or 3 hours without throttling. For which P-state to revisit, we evaluate stepping one P-state above the current upper bound and restoring a previously recorded upper bound (potentially skipping multiple P-states). The resulting policies are:

- **(30min, +1):** P-state schedules include revisits each 30 minutes without throttling. At each revisit, the schedule steps to one P-state above the current upper bound.
- **(3h, +1):** P-state schedules include revisits each 3 hours without throttling. At each revisit, the schedule steps to one P-state above the current upper bound.
- **(3h, restore):** P-state schedules include revisits each 3 hours without throttling. At each revisit, the schedule restores a previously recorded upper bound if the current upper bound is below it; otherwise, it steps to one P-state above the current upper bound.

In all cases, if throttling occurs after a revisit, the resulting epoch transition restores the upper bound that was in effect before the revisit.

Heterogeneous processors. For heterogeneous processors (e.g., our Intel i9-13900), where P-cores support more P-states than E-cores, our prototype updates the scheduled P-state of the P-cores and E-cores by the same number of P-state steps if possible. If, during convergence, the E-core P-state has already reached its minimum or maximum supported value, our implementation continues adjusting only the P-core P-state. Other implementations are also possible.

8. Evaluation

Experimental setup. For most experiments, we focus on four representative machines: two with client-class processors (Intel Core i9-9900 and AMD Ryzen 7 8700G) and two with server-class processors (Intel Xeon Gold 6548Y+ and AMD EPYC 7763). We also run a subset of our experiments on additional machines. All machines except for the two representative client-class ones are part of Chameleon’s and CloudLab’s cloud infrastructures [45], [46]. All machines run Ubuntu 24.04 LTS. We use the default configuration on all machines except for the i9-9900 one, where we disable Hyper-Threading to more closely resemble the setup used in the original Hertzbleed paper [1]. On each machine, we get the minimum supported P-state f_{min} using `cpuinfo_min_freq` and empirically determine the maximum supported P-state f_{max} when all cores are active. $|P|$ is equal to the

11. Some recent AMD processors support a similar interrupt via the AMD HFI driver, but, at the time of writing, this driver is not available on the machines of our experimental setup [44].

TABLE 1. MACHINE SPECIFICATIONS AND MEASURED τ AND p_{SIKE}^* FOR EACH PROCESSOR IN OUR EXPERIMENTAL SETUP. f_{min} AND f_{max} DENOTE THE MINIMUM AND MAXIMUM SUPPORTED CORE FREQUENCIES, $|P|$ IS THE NUMBER OF DISCRETE P-STATES BETWEEN f_{min} AND f_{max} , τ IS THE MAXIMUM THROTTLING REACTION TIME OBSERVED IN OUR MEASUREMENTS, AND p_{SIKE}^* IS THE P-STATE GOLDILOCKS CONVERGES TO FOR SIKE ON EACH MACHINE.

Processor	f_{min} (GHz)	f_{max} (GHz)	$ P $	τ	p_{SIKE}^* (GHz)
Intel Core i9-9900	0.8	4.5	38	3	3.8
Intel Xeon Gold 6548Y+	0.8	3.5	28	4	3.0
AMD Ryzen 7 8700G	0.4	5.075	188	5	4.775
AMD EPYC 7763	0.4	3.25	115	5	2.875
Intel Xeon Gold 6126	1.0	3.3	24	4	3.0
Intel Xeon Gold 6242	1.2	3.5	24	4	3.1
Intel Xeon Gold 6240R	1.0	3.1	22	2	2.9
Intel Xeon Platinum 8380	0.8	3.0	23	1	2.8
Intel Core i9-13900 ¹²	0.8	5.3	46	3	3.0
Intel Xeon CPU Max 9462	0.8	3.1	24	4	2.4

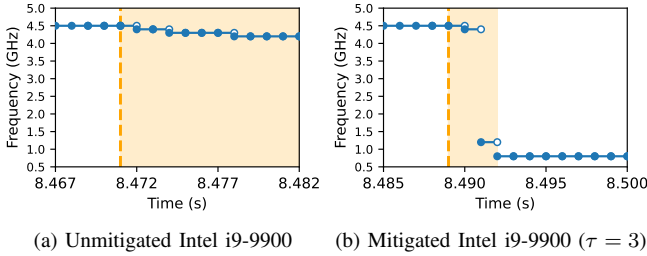


Figure 3. Zoomed-in frequency traces after throttling is detected (dashed vertical line) in Figures 4a and 4b. The shaded orange region indicates when the machine is throttling. For each machine in Table 1, τ is empirically measured as the maximum observed time (across 100 measurements while running SIKE) between when throttling is first detected and when Goldilocks lowers the P-state to f_{min} for cooldown.

number of P-states between f_{min} and f_{max} . We empirically measure the machine-dependent τ (visualized in Figure 3) as the time between when throttling is first detected and when Goldilocks lowers the P-state to f_{min} for cooldown. For each machine, we report the maximum observed τ across 100 measurements (while running SIKE). The cooldown period is fixed to 4 seconds, which we observe is sufficient to start a new epoch without throttling. The characteristics of each machine are summarized in Table 1.

For evaluation, we choose five different cryptographic workloads: three of which are known to be vulnerable to Hertzbleed attacks (SIKE, ECDSA, Classic McEliece [1], [3]), and two standard cryptographic workloads (AES, RSA) from OpenSSL. For each experiment, unless otherwise stated, we set the number of workload threads to $5 \times$ the number of logical cores, so that all cores are fully utilized.

To sample the CPU frequency during experiments, we read `scaling_cur_freq` from the `CPUFreq` subsystem. To measure energy consumption, we use the Running Average Power Limit (RAPL) interface.

12. The reported values in this row correspond to P-cores. The respective values for E-cores are 0.8, 4.2, 35, 3, and 1.9.

8.1. Performance evaluation

We start by evaluating the performance and energy overhead of Goldilocks both before and after it converges to p^* .

8.1.1. Convergence phase. We first compare how different Goldilocks policies behave before converging to p^* . Figure 4 shows example frequency traces on our Intel i9-9900 running SIKE’s decapsulation under four scenarios: no mitigation, linear policy, doubling policy, and binary policy (all static). All three Goldilocks policies eventually converge to the same p^* , but differ in how many epoch transitions they incur and in how they explore P-states within each epoch.

For each policy, we measure the number of epochs required to reach p^* and the average CPU frequency during the first 1000 seconds of execution. This duration was empirically chosen to accommodate slower-converging policies (e.g., binary) while allowing a fair comparison with faster-converging policies that spend more time running at lower frequencies due to more frequent epoch transitions. For example, the linear policy incurs the most epoch transitions and therefore spends more time at low frequencies during convergence (i.e., at f_{min} during cooldown periods). However, because it converges faster, it spends more time at the converged frequency p^* during the initial 1000 seconds.

Figure 5 shows that the binary policy typically achieves the lowest average frequency during the convergence window, as it spends extended time exploring P-states below p^* before moving back up. In contrast, the doubling policy generally results in the highest average frequency, as it can efficiently skip over multiple non-optimal P-states. The only exception is the AMD server machine, where binary converges quickly because p^* happens to coincide with one of the earliest midpoints of the safe P-state range.

These experiments also highlight that p^* is highly machine dependent. Table 1 lists p_{SIKE}^* across the ten processors we evaluate, and the values vary widely. This underscores the importance of letting Goldilocks dynamically determine a “just right” P-state online, rather than relying on static, hand-tuned settings.

A note on throttling and base frequency. While analyzing Table 1, we observe an effect that impacts how p^* should be interpreted on several Intel server processors (Xeon Gold 6548Y+, Xeon Gold 6126, and Xeon CPU Max 9462). On these machines, p_{SIKE}^* lies below the lowest *core* P-state we observe in our unmitigated SIKE runs. For example, on the Xeon Gold 6548Y+, $p_{\text{SIKE}}^* = 3.0$ GHz, while the lowest core P-state we observe in unmitigated SIKE is 3.1 GHz.

Investigating these cases, we find that after the core P-state reaches a certain value (e.g., 3.1 GHz on the Xeon Gold 6548Y+), the processor handles further throttling by lowering the *uncore* frequency (governing the interconnect and last-level cache) instead of (or before) lowering the core P-state further. We validate this interpretation by fixing the uncore frequency to its minimum value and re-running SIKE unmitigated and under Goldilocks; in both cases, the

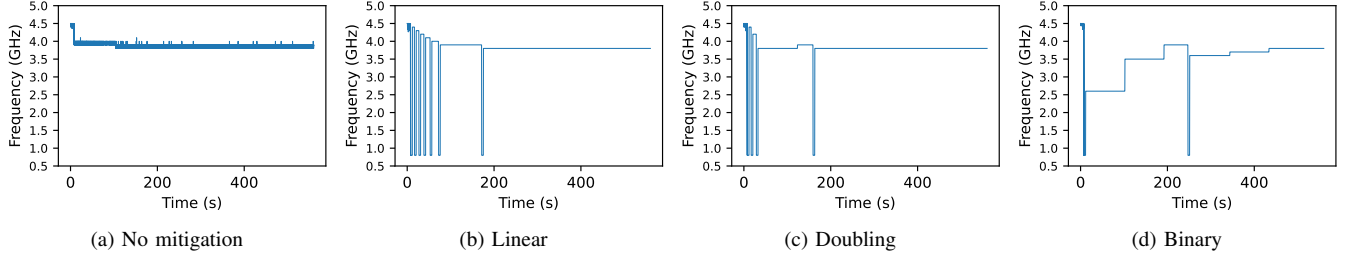


Figure 4. Frequency traces for unmitigated execution and three static Goldilocks policies (linear, doubling, binary) on an Intel Core i9-9900 running SIKE decapsulation. All policies begin at f_{max} and eventually converge to the same p^* , but differ in how they traverse P-states during convergence and in the number of epoch transitions (and cooldown intervals at f_{min}) they incur, leading to different average frequencies during the convergence phase.

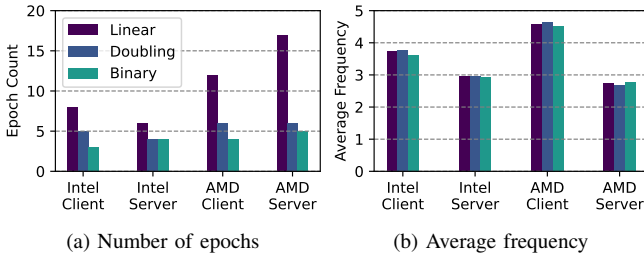


Figure 5. Number of epochs to reach p^* and average CPU frequency during convergence for different static policies. Intel client, Intel server, AMD client, and AMD server correspond to Intel Core i9-9900, Intel Xeon Gold 6548Y+, AMD Ryzen 7 8700G, and AMD EPYC 7763, respectively. Both measurements are computed over the first 1000 seconds of execution. Typically, the binary policy results in the fewest epoch transitions but achieves the lowest average frequency, while the doubling policy results in the highest average frequency during convergence.

core frequency drops to the same lowest P-state. Importantly, we find that uncore frequency still exhibits secret-dependent leakage, making it exploitable under Hertzbleed (Appendix B). Because the throttling interrupt we monitor is raised in these cases as well, on these machines Goldilocks converges to a core p^* that avoids secret-dependent throttling expressed through both core and uncore frequency changes.

Separately, on some Intel server processors and workloads (e.g., SIKE and some of the mixed workloads below), we observe that Goldilocks converges to a p^* that is lower than base frequency. For example, on the Intel Xeon CPU Max 9462, $p_{SIKE}^* = 2.4$ GHz, which is below that processor’s base frequency of 2.7 GHz. This demonstrates that fixing the core frequency to base frequency does not guarantee security: even when the highest allowed P-state is set to the base frequency, throttling can still occur and leak (in this case via the uncore). We did not observe analogous behavior on our AMD machines with our current instrumentation.

8.1.2. Cryptographic workloads. We now evaluate the runtime and energy overheads of Goldilocks on cryptographic workloads. We compare six different configurations:

- 1) **Unmitigated cold:** unmitigated; previously idle.
- 2) **Unmitigated warm:** unmitigated; the same workload has already warmed up the machine, causing throttling.
- 3) **Goldilocks cold:** Goldilocks with linear policy; before convergence; previously idle.

- 4) **Goldilocks warm:** Goldilocks with linear policy; Goldilocks already converged to p^* ; previously idle.
- 5) **Minimum frequency:** the max P-state is fixed to f_{min} , as per prior recommendations [12], [14].
- 6) **Base frequency:** the max P-state is fixed to base frequency, as per prior recommendations [1], [4], [5], [13]

For each machine and workload, we choose the workload size so that it takes approximately 150 seconds to complete in the unmitigated configuration, simplifying comparison.

Table 2 shows the average runtime and energy consumption overheads across different machines and cryptographic workloads. Runtime and energy values are averaged over five runs and normalized to the unmitigated cold configuration for the corresponding machine and workload.

Runtime overhead. Overall, Goldilocks *warm* (after convergence to p^*) incurs low performance overhead compared to the unmitigated baseline and clearly outperforms the fixed-frequency configurations. Across the four main machines and five cryptographic workloads, the average slowdown is 2.75% relative to the unmitigated cold configuration and 2.04% relative to the unmitigated warm configuration. At the same time, the average speed-ups are 21.85 and 495 percentage points relative to the base and minimum frequency configurations, respectively. Intuitively, this is because the cryptographic workloads tend to oscillate among only a few adjacent P-states during unmitigated throttling, and the p^* identified by Goldilocks typically corresponds to the lowest P-state seen in the unmitigated trace. As a result, Goldilocks runs near the maximum sustainable frequency rather than imposing an overly conservative cap. Moreover, in cases where the workload never throttles even without mitigation (for example, AES on the Intel server machine), Goldilocks imposes zero runtime overhead, since no frequency restriction is needed to prevent throttling.

In the Goldilocks *cold* configuration, the runtime overhead is generally higher. Here, the system briefly runs at higher frequencies, triggers throttling, and then spends 4 seconds per epoch at f_{min} (for the cooldown period). This effect is especially pronounced on AMD machines across all workloads due to their finer-grained P-state transitions. That said, this early-phase behavior only matters for short workloads; for longer-running workloads, the amortized overhead approaches that of the Goldilocks warm configuration.

TABLE 2. AVERAGE RUNTIME AND ENERGY OVERHEADS FOR CRYPTOGRAPHIC WORKLOADS ACROSS PROCESSORS AND CONFIGURATIONS. ALL NUMBERS ARE RELATIVE TO THE UNMITIGATED COLD CONFIGURATION FOR THE CORRESPONDING MACHINE AND WORKLOAD. OVERALL, GOLDDLOCKS WARM STAYS CLOSE TO THE UNMITIGATED BASELINE WHILE OUTPERFORMING THE FIXED-FREQUENCY BASELINES.

Processor	Configuration	SIKE		Classic McEliece		ECDSA		RSA-2048		AES-256	
		Runtime Overhead	Energy Overhead	Runtime Overhead	Energy Overhead	Runtime Overhead	Energy Overhead	Runtime Overhead	Energy Overhead	Runtime Overhead	Energy Overhead
Intel Core i9-9900	Unmitigated Cold	-	-	-	-	-	-	-	-	-	-
	Unmitigated Warm	+1%	-1%	+1%	-2%	+1%	-1%	+2%	-1%	+2%	+1%
	Goldilocks Cold	+12%	0%	+16%	0%	+11%	-1%	+16%	+1%	+12%	-1%
	Goldilocks Warm	+3%	-10%	+3%	-7%	+2%	-5%	+4%	-5%	+4%	-4%
	Base Frequency	+28%	-30%	+20%	-23%	+25%	-28%	+24%	-26%	+45%	-34%
	Min. Frequency	+398%	-59%	+370%	-58%	+385%	-62%	+381%	-60%	+397%	-68%
Intel Xeon Gold 6548Y+	Unmitigated Cold	-	-	-	-	-	-	-	-	-	-
	Unmitigated Warm	+1%	+1%	+1%	+5%	0%	0%	0%	0%	+1%	+1%
	Goldilocks Cold	+14%	+6%	+7%	+2%	+13%	-1%	+22%	+8%	+1%	-2%
	Goldilocks Warm	+4%	+6%	0%	-3%	+4%	-2%	+5%	+7%	0%	+1%
	Base Frequency	+26%	+10%	+3%	-1%	+25%	+2%	+9%	+10%	+59%	+26%
	Min. Frequency	+295%	+105%	+209%	+64%	+293%	+92%	+244%	+84%	+478%	+241%
AMD Ryzen 7 8700G	Unmitigated Cold	-	-	-	-	-	-	-	-	-	-
	Unmitigated Warm	+1%	-1%	+1%	-2%	+1%	-3%	0%	+1%	0%	-2%
	Goldilocks Cold	+22%	-1%	+49%	-5%	+25%	+1%	+58%	-2%	+2%	-1%
	Goldilocks Warm	+3%	-12%	+2%	-7%	+2%	-11%	+3%	-11%	+1%	-6%
	Base Frequency	+14%	-37%	+10%	-29%	+14%	-36%	+7%	-20%	+19%	-41%
	Min. Frequency	+771%	-4%	+744%	-9%	+774%	-4%	+722%	-9%	+815%	+58%
AMD EPYC 7763	Unmitigated Cold	-	-	-	-	-	-	-	-	-	-
	Unmitigated Warm	0%	0%	0%	0%	0%	0%	0%	0%	+1%	+1%
	Goldilocks Cold	+38%	+7%	+1%	+1%	+35%	+3%	+32%	+7%	+2%	+1%
	Goldilocks Warm	+6%	+2%	0%	+1%	+3%	-2%	+6%	-1%	+1%	+1%
	Base Frequency	+25%	-11%	+18%	-17%	+21%	-11%	+26%	-11%	+31%	+4%
	Min. Frequency	+673%	+152%	+388%	+96%	+652%	+125%	+680%	+135%	+639%	+252%

Energy overhead. In terms of energy overhead, we observe that Goldilocks (both warm and cold) can sometimes improve energy efficiency relative to the unmitigated baseline. On client-class machines, Goldilocks warm achieves an average 7.3% reduction in total energy consumption, with reductions ranging from 4-12%. This occurs because the lower power consumption at the reduced frequency more than compensates for the slightly longer runtime. On server-class machines, the picture is more mixed: Goldilocks warm introduces an average energy overhead of 0.7%, with some workloads seeing modest savings and others modest overhead. In fact, in our experiments, minimum frequency consistently consumes the most energy on server machines. This reflects differences in processor design: server processors are optimized for energy efficiency, so the energy cost of running longer at a reduced frequency may outweigh any power savings. Client machines, which typically prioritize performance, on average, save energy when running slower.

8.1.3. Mixed workloads. Having evaluated cryptographic workloads in isolation, we now measure the overhead of Goldilocks for cryptographic workloads running simultaneously with other workloads. Specifically, we measure the performance of SIKE under Goldilocks while workloads from the SPEC benchmark suite execute in the background. We chose SIKE as our representative cryptographic workload because it is vulnerable to Hertzbleed and is compute-intensive like other cryptosystems. The machine’s physical cores are split evenly: one half runs SIKE decapsulation with $2.5\times$ the logical core count in threads, and the other

half runs one SPECrate benchmark per logical core. We use the same SIKE workload size as in Section 8.1.2, resulting in a similar 150-second execution time (for SIKE) in the unmitigated configuration across all machines. We test each SPECrate benchmark individually using the reference input size, producing 23 scenarios where SIKE runs alongside a different background workload. This setup reflects noisy system conditions where cryptographic operations execute concurrently with processes that vary in power consumption. The different thread counts between SIKE and SPECrate are due to SPECrate’s substantial memory requirements, which limit how many SPEC copies can run at once.

We focus on the following configurations: unmitigated cold, Goldilocks cold, Goldilocks warm, and base frequency. For all Goldilocks configurations, we use the linear static policy. Based on Section 8.1.2, we exclude unmitigated warm because it performs similarly to unmitigated cold, with at most a 1% runtime overhead. We also exclude the minimum frequency configuration because its substantial overhead is impractical despite guaranteeing security. Runtime and energy consumption values are measured by running SIKE for a fixed number of iterations, while SPEC executes on other cores. The SPEC workload is terminated once SIKE completes. Overheads are computed for each of the 23 SIKE+SPECrate mixes relative to the unmitigated cold baseline and then averaged across all mixes.

Runtime and energy overheads. Across all machines, Goldilocks warm incurs higher overhead on SIKE in the presence of background workloads than when SIKE runs in

TABLE 3. AVERAGE RUNTIME AND ENERGY OVERHEADS FOR MIXED WORKLOADS (SIKE CO-RUNNING WITH SPEC RATE) ACROSS PROCESSORS AND CONFIGURATIONS. ALL NUMBERS ARE RELATIVE TO THE UNMITIGATED COLD CONFIGURATION FOR THE CORRESPONDING MACHINE. OVERALL, GOLDDLOCKS WARM REMAINS FASTER THAN FIXED-FREQUENCY BASELINES BUT INCURS HIGHER OVERHEAD THAN WHEN SIKE RUNS IN ISOLATION.

Processor	Configuration	Runtime Overhead	Energy Overhead
Intel Core i9-9900	Unmitigated Cold	-	-
	Goldilocks Cold	+13%	-9%
	Goldilocks Warm	+9%	-23%
	Base Frequency	+25%	-34%
Intel Xeon Gold 6548Y+	Unmitigated Cold	-	-
	Goldilocks Cold	+24%	+11%
	Goldilocks Warm	+15%	+6%
	Base Frequency	+22%	-6%
AMD Ryzen 7 8700G	Unmitigated Cold	-	-
	Goldilocks Cold	+30%	-2%
	Goldilocks Warm	+3%	-12%
	Base Frequency	+13%	-32%
AMD EPYC 7763	Unmitigated Cold	-	-
	Goldilocks Cold	+40%	+8%
	Goldilocks Warm	+9%	+5%
	Base Frequency	+22%	-6%

isolation. For example, relative to the unmitigated cold baseline, SIKE experiences an average overhead of 9.01% under Goldilocks warm with SPEC running in the background (Table 3), compared to 4.5% when running alone (Table 2). Under mixed workloads, Goldilocks warm continues to provide a performance improvement over the base frequency configuration; however, this improvement is smaller than when SIKE runs in isolation, resulting in an average 11.69 percentage point difference across all machines. This is likely due to the increased variability in power consumption introduced by SPEC. Without a background workload, the lowest P-state observed in the unmitigated SIKE execution is close to the average P-state, reflecting SIKE’s uniform power consumption. With SPEC, the lowest P-state observed can be much lower than the average P-state because SPEC exhibits more variable power consumption than SIKE. Accordingly, Goldilocks must choose a p^* that avoids throttling even during peak background activity (and on the Xeon Gold 6548Y+, this includes avoiding uncore-frequency throttling; see the note in Section 8.1.1), resulting in higher overhead.

Similarly, Table 3 shows Goldilocks cold incurs higher overhead under mixed workloads (26.75% on average) than under isolated workloads due to the increased time spent at f_{min} during cooldown. Energy efficiency follows the same trend as in the prior section: Goldilocks warm achieves 17.2% average savings on client machines but incurs 5.5% overhead on server machines relative to unmitigated cold.

8.1.4. Changing workloads. The previous sections evaluated Goldilocks under stable conditions with a constant workload. We now consider a changing-headroom scenario in which a transient, power-hungry workload starts and completes in the background during the execution of a long-running cryptographic workload. This transient workload

TABLE 4. OVERHEADS FOR CHANGING WORKLOADS (LONG-RUNNING SIKE WITH A TRANSIENT SPEC RATE BACKGROUND WORKLOAD) ACROSS PROCESSORS AND CONFIGURATIONS. ALL NUMBERS ARE RELATIVE TO THE UNMITIGATED COLD CONFIGURATION FOR THE CORRESPONDING MACHINE. ADAPTIVE REVISIT PATTERNS REDUCE RUNTIME OVERHEAD RELATIVE TO A STATIC POLICY, AT THE COST OF ADDITIONAL EPOCH TRANSITIONS.

Processor	Goldilocks Policy	Runtime Overhead	Energy Overhead	Epoch Count
Intel Core i9-9900	Unmitigated	-	-	-
	Static	+5%	-10%	9
	Adapt. (30min, +1)	+1%	-1%	45
	Adapt. (3h, +1)	+3%	-6%	14
	Adapt. (3h, restore)	+2%	-5%	15
Intel Xeon Gold 6548Y+	Unmitigated	-	-	-
	Static	+16%	+5%	9
	Adapt. (30min, +1)	+8%	+5%	45
	Adapt. (3h, +1)	+10%	+4%	13
	Adapt. (3h, restore)	+9%	+5%	14
AMD Ryzen 7 8700G	Unmitigated	-	-	-
	Static	+7%	-23%	29
	Adapt. (30min, +1)	+1%	-2%	51
	Adapt. (3h, +1)	+6%	-20%	32
	Adapt. (3h, restore)	+1%	-2%	35
AMD EPYC 7763	Unmitigated	-	-	-
	Static	+10%	-5%	24
	Adapt. (30min, +1)	+3%	-1%	52
	Adapt. (3h, +1)	+9%	-5%	26
	Adapt. (3h, restore)	+3%	-1%	29

can temporarily reduce available power headroom, causing Goldilocks to converge to a lower P-state and remain “stuck” below the frequency the cryptographic workload could sustain in isolation after the background workload ends. In this setting, we also evaluate the performance-security trade-off introduced by adaptive policies that recover performance by revisiting P-states that previously caused throttling.

We run SIKE with $5\times$ the logical core count in threads and choose the workload size so that it takes approximately 20 hours to complete in the unmitigated configuration. After the initial 10 minutes, we launch a SPECrate benchmark with half the logical core count copies, unpinned, and let it run to completion. For each machine, we select a SPECrate benchmark that forces Goldilocks to converge to a lower p^* than what SIKE could sustain in isolation. We measure SIKE’s runtime and energy overheads, as well as the number of additional epoch transitions induced by adaptive revisits.

We evaluate Goldilocks cold under a static linear policy (no revisits) and the adaptive policies described in Section 7. For the (3h, restore) policy, each revisit restores the upper bound to a value recorded from an isolated SIKE run (before launching the transient workload). All configurations start from a previously idle state and apply a linear static policy at epoch transitions that are not induced by revisits. The results in this section are normalized to an unmitigated baseline running the same 20h SIKE + transient SPECrate workloads.

Performance recovery vs. additional epoch transitions. Table 4 shows that adaptive policies recover performance relative to a static policy after the transient workload ends, at the cost of additional epoch transitions induced by revisits.

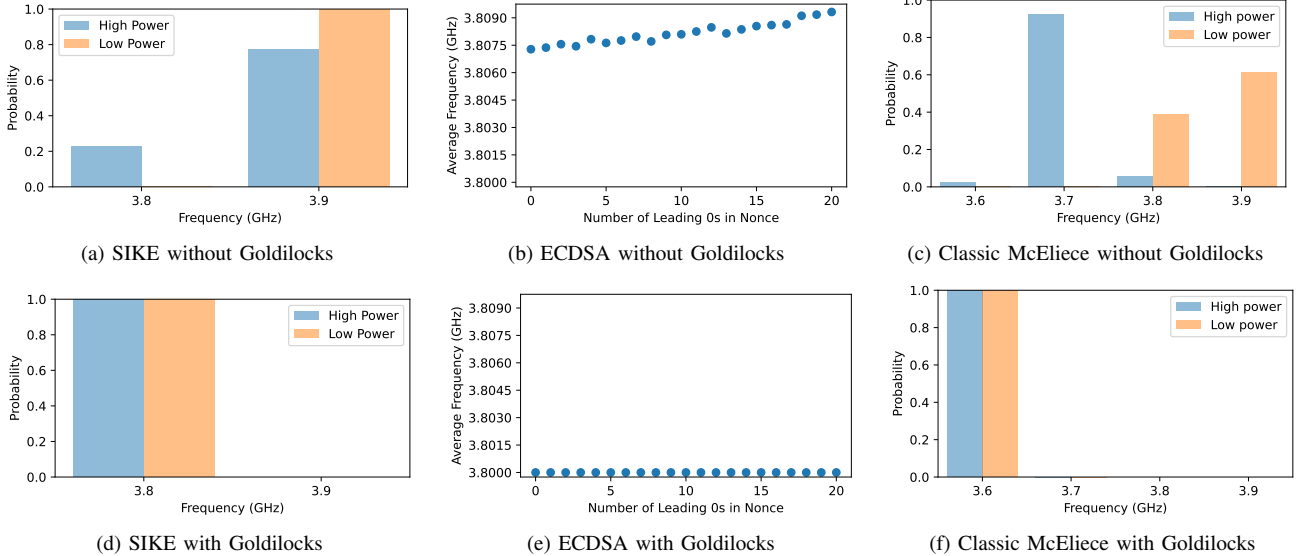


Figure 6. Frequency distributions observed while reproducing published proof-of-concept attacks against SIKE, ECDSA, and Classic McEliece on our Intel Core i9-9900. Without Goldilocks (Figures 6a-6c), CPU frequency is correlated with the secret (e.g., the number of leading zeros in the ECDSA nonce). With Goldilocks cold (Figures 6d-6f), the distributions do not exhibit a secret-dependent separation in our experiments.

Across all machines, among adaptive policies varying only in revisit interval, shorter intervals yield lower overhead: the (30min, +1) policy averages 3.39% runtime overhead, while the (3h, +1) policy averages 7.15% relative to the unmitigated baseline. These are 5.14 and 2.66 percentage points lower than the static policy, respectively. However, this comes at the cost of additional epoch transitions: when averaged across the four machines, the (30min, +1) and (3h, +1) policies incur 31 and 4 more epoch transitions, respectively, than the static policy, which performs no revisits.

The impact of revisit intervals differs between Intel and AMD processors due to AMD’s smaller P-state step size (25 MHz vs. 100 MHz on Intel), even though the gap between p_{SIKE}^* and the lowest P-state scheduled by Goldilocks is similar on both platforms. AMD processors benefit more from shorter revisit intervals, achieving a larger reduction in runtime overhead (5.68 vs. 1.85 percentage points on Intel) when moving from the (3h, +1) to the (30min, +1) policy.

The choice of which higher P-state to revisit also affects the performance-security trade-off. With the same 3-hour revisit interval, the (3h, restore) policy reduces average runtime overhead to 4.02%, improving over (3h, +1) by 3.13 percentage points while introducing a similar number of additional epoch transitions. Compared to (30min, +1), (3h, restore) achieves comparable runtime overhead with substantially fewer additional epoch transitions. These results show that “restore”-style revisits can improve the performance-security trade-off by recovering performance more quickly without increasing the frequency of revisits.

8.2. Security evaluation

We now evaluate the security of Goldilocks against Hertzbleed-vulnerable cryptographic workloads by repro-

ducing previously published attacks. We focus on the proof-of-concept attacks against SIKE, Classic McEliece, and ECDSA from prior work [1], [3], modeling a local adversary who can observe the current CPU frequency every 1 ms (i.e., once per time quantum on our processors). We refer to the original papers for details on the cryptanalysis.

Attack setup. We consider the following three configurations: unmitigated warm, Goldilocks cold (with the linear static policy), and Goldilocks warm. Unmitigated warm demonstrates the workload’s baseline vulnerability. Goldilocks cold and warm evaluate how well Goldilocks mitigates the attacks before and after convergence to p^* .

Each attack follows the setup of its original work. For SIKE, we adopt the Cloudflare CIRCL attack setup [1]. We spawn 300 concurrent threads that continuously issue decapsulation requests that induce key bit-dependent power consumption for 200 seconds. We repeat this for 10 randomly generated keys, attacking 4 of the 378 key bits. For Classic McEliece, we use the round-4 NIST submission and exploit the decoding-failure vulnerability described in prior work [3], which causes secret-dependent power consumption. We run the decapsulation routine on all cores for 100 seconds for each of the 9 input ciphertexts. For ECDSA, we reproduce the BearSSL attack where power consumption varies with the number of leading zero bits in the nonce [3]. We execute repeated signing requests on all cores for 120 seconds and, for each configuration, test multiple nonces with the same number of leading zeros. In all experiments, we sample the CPU frequency every 1 ms. For each target, we run Goldilocks continuously for the entire experiment: the Goldilocks cold configuration starts unconverted only at the beginning of that target’s experiment; after it converges to p^* , all subsequent invocations execute in the warm state.

Attack results. In the unmitigated warm configuration, the published proof-of-concept attacks succeed (Figures 6a-6c), confirming that these workloads exhibit an exploitable secret-dependent CPU frequency signal under our setup.

In the Goldilocks cold configuration, throttling-induced frequency variations occur during the convergence phase, before Goldilocks reaches p^* . However, in our experiments, these variations are too short: convergence always completes within 30 epoch transitions, and the cumulative throttling time is never greater than 150 ms (in the worst case, observed on our AMD EPYC 7763 processor). As a result, the published proof-of-concept attacks fail to recover meaningful secret information (e.g., see Figures 6d-6f for the results on our i9-9900 processor). That said, we do not claim that Goldilocks cold eliminates leakage entirely: an optimal adversary could, in principle, attempt to exploit this finite number of epoch transitions to leak secret information. Section 6 provides a conservative worst-case bound on what can be inferred from the resulting P-state sequences.

In the Goldilocks warm configuration, the proof-of-concept attacks also fail. Empirically, the attacker observes no secret-dependent frequency variations in the warm state. That is, once Goldilocks has converged to p^* , the observed P-state sequence remains constant in our experiments. In summary, in our experiments Goldilocks removes the secret-dependent frequency signal exploited by these published Hertzbleed proof-of-concept attacks.

Implications for adaptive policies. Adaptive policies warrant separate discussion because revisits can induce additional epoch transitions beyond those needed for convergence, and over long executions, the number of such transitions may be unbounded. Specifically, whenever a P-state schedule includes a revisit above the current upper bound, the victim’s power consumption can again influence whether throttling occurs during that revisit. In practice, this means adaptive policies should use revisit patterns that are sufficiently sparse on the timescale of the expected attack horizon, so that only a few additional epoch transitions (and thus leakage) arise. We discuss concrete configuration guidance for adaptive policies in Section 9.

9. Discussion

Adversary observability. Our leakage analysis assumes a worst-case adversary who can observe the CPU P-state at every time quantum during the victim’s execution. In practice, attackers often have weaker observability: they may only see coarse-grained frequency information (e.g., average frequency over a time window) or end-to-end response times from remote requests. From an information-theoretic perspective, such coarsening of observations can only *reduce* the channel capacity. Intuitively, mapping a length- t P-state sequence to a smaller set of aggregate statistics (such as average frequency over t_k time quanta) merges multiple distinct P-state sequences into the same observed value and therefore cannot increase the number of distinguishable outcomes. In Appendix A, we formalize this intuition by

showing that, for any fixed window length t_k , the number of distinct average-frequency observations is at most the number of distinct P-state sequences of length t_k , and the same holds when the observations are end-to-end response times. Thus, any adversary that infers frequency only through coarse-grained measurements or remote timings can leak no more information than the worst-case adversary we analyze.

Beyond core P-states. Although we present Goldilocks in terms of CPU core P-states, the underlying analysis should apply broadly to settings where power management mechanisms produce an observable trace and the defender can restrict which traces are realizable. Our evaluation touches one such case: on some Intel server processors, throttling is sometimes manifested through changes in the uncore frequency domain, and Goldilocks still converges to a core P-state that avoids secret-dependent throttling behavior. Future work could extend Goldilocks to also control uncore frequency, potentially improving performance on such processors. More generally, “P-state” in our analysis can be interpreted as a vector of power management decisions, not just a scalar CPU frequency. For example, mechanisms that schedule work across cores based on power consumption [5] can leak information via which cores are active in addition to their frequencies. Extending Goldilocks to treat the combination of per-core frequency and core-activity patterns as the state space would expand the policy design space while preserving the same style of leakage accounting.

Performance degradation attacks. Goldilocks bounds leakage under an adversary who can observe all P-states the CPU runs at during the victim’s execution (Section 3). If co-located with the victim, such an adversary may also be able to influence the system’s power consumption by running workloads in parallel with the victim. This may force Goldilocks to adopt more conservative P-state schedules and remain “stuck” at low P-states for the remainder of the victim’s execution (or until a revisit), degrading performance. That said, an attacker who can run arbitrary concurrent code can degrade a victim’s performance even without any help from Goldilocks, for example via resource contention or scheduler behaviors [5], [47]–[49]. Thus, while mitigating performance degradation attacks is an important problem, addressing it is orthogonal to the aims of this work.

Responsiveness under changing conditions. Goldilocks works best when the victim runs under relatively stable power headroom, in which case it can converge to a “just right” P-state that avoids throttling while preserving performance. In some deployments, however, available power headroom may change over time (e.g., due to ambient conditions or co-scheduled workloads), and p^* may increase or decrease accordingly. A key limitation of Goldilocks is responsiveness to such changes. As Section 8.1.4 shows, in these cases, static policies can leave performance on the table after converging to a conservative p^* , whereas adaptive policies can recover performance by revisiting higher P-states. This recovery induces an inherent trade-off: making

revisits frequent improves responsiveness but also creates more opportunities for leakage, whereas making revisits sparse limits leakage but reduces responsiveness.

One interesting direction for improving responsiveness under changing headroom may be to allow secret-independent system signals to influence P-state choices *within* an epoch. For example, Goldilocks could conservatively lower P-states when resource-intensive background workloads start and restore them when those workloads end, aiming to avoid throttling altogether. Realizing this extension would require new monitoring mechanisms and additional analysis beyond Section 6, but it could substantially improve responsiveness in dynamic environments.

Choosing the policy. Deploying Goldilocks requires choosing a policy that governs P-state schedules. In Section 8, we evaluate a few representative choices (both static and adaptive). However, more aggressive adaptive policies (with more frequent revisits) are also possible. One practical way to configure such policies is to start from a time window T that reflects the attack duration the operator cares to defend against and then choose policy parameters so that the maximum number of epoch transitions within T stays below what known Hertzbleed receivers (against the target cryptosystem) require under comparable conditions. For example, consider a SIKE-like victim where a key bit-dependent power difference could shift the highest non-throttling P-state by a single step (cf. Figure 6a). In an idealized model, it may be possible to construct a receiver that leaks (in expectation) 2 key bits per revisit by probing key bits one at a time and observing whether throttling (and the associated epoch transition) occurs again.¹³ Thus, an operator who wants to limit how many key bits leak over T should allow only a correspondingly small number of revisits within T . Note that this guidance complements Section 6: the leakage bounds we derive there are cryptosystem- and receiver-agnostic, while this framing lets operators tune policies using known receivers for a specific cryptosystem.

Composability with other mitigations. Goldilocks provides an information-theoretic upper bound on the amount of information that Hertzbleed can leak through frequency throttling as a function of execution time and policy parameters. This makes it complementary to other mitigations. For example, hardware vendors have recommended frequent key refreshes as a way to limit long-term leakage of cryptographic keys via Hertzbleed [8], [9]. Given a target leakage budget L^* and machine-specific parameters, our analysis can inform how often keys must be refreshed so that the maximum leakage per key stays below L^* . Similarly, masking and other implementation-level countermeasures reduce the amount of secret-dependent power variation per operation, and rate-limiting mechanisms reduce the number of observations available to a remote attacker. Goldilocks

13. If each tested bit independently triggers throttling with probability $1/2$ and the revisit ends at the first such bit (because throttling triggers an epoch transition), then the expected number of tested bits per revisit follows a geometric distribution with mean 2.

can be deployed alongside these techniques, with its leakage bounds interpreted as applying to a channel whose per-sample leakage has already been reduced. In this way, Goldilocks can serve as a conservative upper bound on leakage, on top of which defenders can layer more specialized mitigations to meet application-specific goals.

10. Related work

This work builds on a rich line of research that uses quantitative information flow to model and bound leakage through side channels [29]–[34]. In this framework, a side channel is modeled as a channel from secrets to observations, and information-theoretic notions (such as Shannon entropy) are used to derive worst-case leakage bounds and to guide the design of mitigations. These ideas have been applied to timing channels [32]–[34] and microarchitectural side channels [30], [31]. Closest in spirit to our work are approaches that enforce quantitative bounds by monitoring executions and dynamically shaping observable behavior, such as Askarov et al.’s predictive mitigation of timing channels [32]. Goldilocks applies this quantitative perspective to a different kind of channel: the Hertzbleed side channel induced by CPU throttling. Instead of padding time directly, Goldilocks monitors throttling, restricts which P-state sequences can arise, and uses an information-theoretic model of those sequences to derive an $O(N \log t)$ worst-case leakage bound. To our knowledge, this is the first system that combines a practical, OS-level mitigation for Hertzbleed-style attacks with a formal, asymptotic leakage guarantee.

11. Conclusion

In this paper, we modeled Hertzbleed as an information-theoretic channel and introduced Goldilocks, a framework that lets defenders constrain how much information the channel can carry. We demonstrated that Goldilocks can be implemented using existing processor interfaces without changes to application software, provides formal leakage bounds, and keeps performance close to an unmitigated system. The net result is a practical Hertzbleed mitigation that substantially improves on prior approaches.

Acknowledgment

We thank the reviewers for their helpful feedback. We also thank the authors of the original Hertzbleed paper [1] for many helpful conversations and Timothy Sherwood for his insightful questions about Hertzbleed that helped spark this work. This research was supported in part by ARL grant W911NF-25-1-0179, a CyLab Seed grant, a Pennsylvania Infrastructure Technology Alliance grant, and a Korea Foundation for Advanced Studies fellowship.

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] C. Liu, A. Chakraborty, N. Chawla, and N. Roggel, "Frequency throttling side-channel attack," in *CCS*, 2022.
- [3] Y. Wang, R. Paccagnella, A. Wandke, Z. Gang, G. Garrett-Grossman, C. W. Fletcher, D. Kohlbrenner, and H. Shacham, "DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data," in *S&P*, 2023.
- [4] 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," in *USENIX Security*, 2023.
- [5] I. Chun, I. Siu, and R. Paccagnella, "Scheduled disclosure: Turning power into timing without frequency scaling," in *S&P*, 2025.
- [6] T. Yu, C. Cheng, Z. Yang, Y. Wang, Y. Pan, and J. Weng, "Hints from hertz: Dynamic frequency scaling side-channel analysis of number theoretic transform in lattice-based KEMs," *CHES*, vol. 2024, 2024.
- [7] Intel, "INTEL-SA-00698," <https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00698.html>, 2022, accessed on Sep 7, 2025.
- [8] —, "Frequency throttling side channel software guidance for cryptography implementations," <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/frequency-throttling-side-channel-guidance.html>, 2022, accessed on Sep 7, 2025.
- [9] AMD, "Frequency scaling timing power side-channels," <https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1038.html>, 2022, accessed on Sep 7, 2025.
- [10] Arm, "Arm CPU security update: Power-management throttling side-channel," <https://developer.arm.com/documentation/110396/1-1>, 2022, accessed on Nov 13, 2025.
- [11] Ampere, "Hertzbleed - bulletin AMP-SB-0005," <https://amperecomputing.com/products/security-bulletins/hertzbleed>, 2022, accessed on Nov 13, 2025.
- [12] PQC Email List, "HertzBleed : power side channel attacks on SIKE," https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/jAj_8Hreqc0, 2022, accessed on Nov 13, 2025.
- [13] D. J. Bernstein, "Timing attacks: Overclocking FAQ," <https://timing.attacks.cr.yt.to/overclocking.html>, 2022, accessed on Nov 13, 2025.
- [14] —, "cr.yt.to: 2023.06.09: Turbo Boost," <https://blog.cr.yt.to/2023/0609-turboboost.html>, 2023, accessed on Nov 13, 2025.
- [15] S. Mangard, N. Pramstaller, and E. Oswald, "Successfully attacking masked AES hardware implementations," in *CHES*, 2005.
- [16] S. Bauer, "Attacking exponent blinding in RSA without CRT," in *COSADE*, 2012.
- [17] W. Schindler and A. Wiemers, "Power attacks in the presence of exponent blinding," *J. Cryptogr. Eng.*, vol. 4, no. 4, 2014.
- [18] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert, "On the cost of lazy engineering for masked software implementations," in *CARDIS*, 2014.
- [19] W. Schindler and A. Wiemers, "Generic power attacks on RSA with CRT and exponent blinding: new results," *J. Cryptogr. Eng.*, vol. 7, no. 4, 2017.
- [20] K. Papagiannopoulos and N. Veshchikov, "Mind the gap: Towards secure 1st-order masking in software," in *COSADE*, 2017.
- [21] S. Gao, B. Marshall, D. Page, and E. Oswald, "Share-slicing: Friend or foe?" *TCHES*, 2020.
- [22] K. Ngo, E. Dubrova, Q. Guo, and T. Johansson, "A side-channel attack on a masked IND-CCA secure saber KEM," *TCHES*, 2021.
- [23] A. Beckers, L. Wouters, B. Gierlichs, B. Preneel, and I. Verbauwhede, "Provable secure software masking in the real-world," in *COSADE*, 2022.
- [24] E. Barker, "NIST special publication 800-57 part 1 rev. 5, recommendation for key management," 2020.
- [25] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2008, vol. 31.
- [26] L. Wu and S. Picek, "Remove some noise: On pre-processing of side-channel measurements with autoencoders," *TCHES*, 2020.
- [27] M. Randolph and W. Diehl, "Power side-channel attack analysis: A review of 20 years of study for the layman," *Cryptography*, vol. 4, no. 2, 2020.
- [28] O. Bronchain and F.-X. Standaert, "Side-channel countermeasures' dissection and the limits of closed source security evaluations," *TCHES*, 2020.
- [29] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *S&P*, 2009.
- [30] Y. Yuan, Z. Liu, and S. Wang, "CacheQL: Quantifying and localizing cache Side-Channel vulnerabilities in production software," in *USENIX Security*, 2023.
- [31] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Untangle: A principled framework to design low-leakage, high-performance dynamic partitioning schemes," in *ASPLOS*, 2023.
- [32] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *CCS*, 2010.
- [33] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *CCS*, 2011.
- [34] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *CCS*, 2007.
- [35] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2025.
- [36] AMD, *AMD Ryzen Master Overclocking User's Guide for AMD Ryzen and Ryzen Threadripper Processors*, February 2018.
- [37] R. J. Wysocki, "intel_pstate CPU performance scaling driver," https://docs.kernel.org/admin-guide/pm/intel_pstate.html, accessed on Nov 13, 2025.
- [38] H. Rui, "amd-pstate CPU performance scaling driver," <https://docs.kernel.org/admin-guide/pm/amd-pstate.html>, accessed on Oct 23, 2025.
- [39] Intel, "What is clock speed?" <https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html>, accessed on Oct 23, 2025.
- [40] R. J. Wysocki, "Cpu performance scaling," <https://docs.kernel.org/admin-guide/pm/cpufreq.html>, accessed on Oct 23, 2025.
- [41] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, "Collide+Power: Leaking inaccessible data with software-based power side channels," in *USENIX Security*, 2023.
- [42] M. Kalmbach, M. Gottschlag, T. Schmidt, and F. Belloso, "TurboCC: A practical frequency-based covert channel with Intel Turbo Boost," 2020, preprint, arXiv:2007.07046 [cs.CR].
- [43] "Hardware-feedback interface for scheduling on intel hardware," <https://docs.kernel.org/arch/x86/intel-hfi.html>, accessed on Oct 23, 2025.
- [44] P. Yuan and M. Limonciello, "Hardware feedback interface for hetero core scheduling on AMD platform," <https://docs.kernel.org/arch/x86/amd-hfi.html>, accessed on Nov 13, 2025.
- [45] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the Chameleon testbed," in *ATC*, 2020.
- [46] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *ATC*, 2019.
- [47] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *S&P*, 2011.

- [48] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom, “Amplifying side channels through performance degradation,” in *CCS*, 2016.
- [49] Y. Zhu, B. Chen, Z. N. Zhao, and C. W. Fletcher, “Controlled preemption: Amplifying side-channel attacks from userspace,” in *ASPLOS*, 2025.

Appendix A. Adversary observability

We now discuss the worst-case information leakage considering an attacker who can either measure CPU frequency at a coarse granularity or use remote timing analysis to infer the CPU frequency. We assume these attackers can measure the average frequency of t_k time quanta for some $t_k \in \mathbb{Z}^+$, instead of observing the entire P-state sequence.

Let $f_1, f_2, \dots, f_{|P|}$ be the possible CPU frequencies corresponding to the $|P|$ P-states. Then, the average frequency of a single measurement is a value $\sum_{i=1}^{|P|} x_i f_i / t_k$ where x_i is a non-negative integer representing the number of time quanta within t_k spent in frequency f_i such that $\sum_{i=1}^{|P|} x_i = t_k$. The number of observable average frequencies in t_k is limited by the number of different combinations of $x_1, x_2, \dots, x_{|P|}$, and thus the number of different observable average frequencies is at most $\binom{t_k + |P| - 1}{|P| - 1}$.

Using the fact that $\frac{i|P|}{|P|+i-1} \geq 1$ for any $i \in \mathbb{Z}^+$,¹⁴ we show that the number of observable average frequencies in t_k , $\binom{t_k + |P| - 1}{|P| - 1}$, is no greater than the number of realizable P-state sequences within t_k , $|P|^{t_k}$:

$$\begin{aligned} \frac{|P|^{t_k}}{\binom{t_k + |P| - 1}{|P| - 1}} &= \frac{|P|^{t_k} (|P| - 1)! t_k!}{(t_k + |P| - 1)!} \\ &= \frac{|P|^{t_k} t_k!}{(t_k + |P| - 1) \cdots |P|} = \prod_{i=1}^{t_k} \frac{i|P|}{|P| + i - 1} \geq 1. \end{aligned} \quad (6)$$

We first consider an attacker who can measure the average CPU frequency at a coarse granularity, once every t_k time quanta. In time $t = nt_k$, where the attacker can collect n samples of the average frequency each t_k time quanta, the worst-case information leakage is bounded by:

$$\log \left(\frac{t_k + |P| - 1}{|P| - 1} \right)^n = \log \left(\frac{t_k + |P| - 1}{|P| - 1} \right)^{\frac{t}{t_k}}.$$

Using Equation 6, we compare the worst-case information leakage from this attacker to that of an attacker who observes the entire P-state sequence for all t time quanta:

$$\begin{aligned} \log |P|^t - \log \left(\frac{t_k + |P| - 1}{|P| - 1} \right)^{\frac{t}{t_k}} \\ = \log \left(\frac{|P|^t}{\left(\frac{t_k + |P| - 1}{|P| - 1} \right)^{\frac{t}{t_k}}} \right) = \frac{t}{t_k} \log \left(\frac{|P|^{t_k}}{\binom{t_k + |P| - 1}{|P| - 1}} \right) \geq 0. \end{aligned}$$

¹⁴ $i|P| \geq |P| + i - 1$ since $i|P| - (|P| + i - 1) = (|P| - 1)(i - 1) \geq 0$. $\frac{i|P|}{|P| + i - 1} \geq 1$ by dividing both sides by $|P| + i - 1$, since $|P| + i - 1 > 0$.

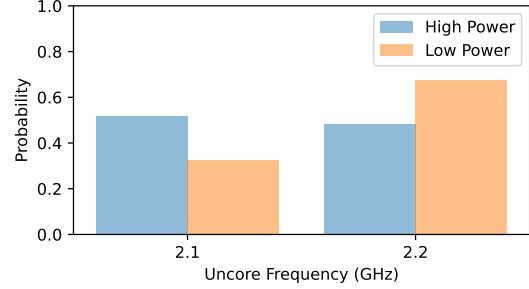


Figure 7. Uncore frequency observed while reproducing the published proof-of-concept attack against SIKE (see Section 8.2) on an unmitigated Intel Xeon Gold 6548Y+, demonstrating secret-dependent leakage.

This shows that an attacker who measures CPU frequency at a coarse granularity leaks information no greater than an attacker who observes the entire P-state sequence.

Next, we consider an attacker who sends n requests and measures the response times to infer the CPU frequency via remote timing analysis. Let $t_1, t_2, \dots, t_n \in \mathbb{Z}^+$ be the response times in time quanta for each of the n requests, where $\sum_{i=1}^n t_i = t$. We conservatively assume that the attacker can infer the average frequency within t_1, t_2, \dots, t_n for each request. The worst-case information leakage that an attacker using remote timing analysis leaks is bounded by:

$$\log \left(\prod_{i=1}^n \binom{t_i + |P| - 1}{|P| - 1} \right).$$

Using Equation 6, we compare the worst-case information leakage from this attacker to that of an attacker who observes the entire P-state sequence for all t time quanta:

$$\begin{aligned} \log |P|^t - \log \left(\prod_{i=1}^n \binom{t_i + |P| - 1}{|P| - 1} \right) \\ = \log \left(\frac{|P|^t}{\prod_{i=1}^n \binom{t_i + |P| - 1}{|P| - 1}} \right) = \sum_{i=1}^n \log \left(\frac{|P|^{t_i}}{\binom{t_i + |P| - 1}{|P| - 1}} \right) \geq 0. \end{aligned}$$

This shows that an attacker who uses remote timing analysis leaks information no greater than an attacker who observes the entire P-state sequence.

Appendix B. Uncore frequency leakage

Recall from Section 8.1.1 that on several Intel server processors, once the core P-state reaches a certain value, the processor handles further throttling by lowering the uncore frequency instead of (or before) lowering the core P-state further. Notably, the uncore frequency remains secret-dependent under Hertzbleed-vulnerable workloads. Figure 7 illustrates this effect for SIKE on the Intel Xeon Gold 6548Y+ processor: under the unmitigated warm configuration, the workload produces a secret-dependent uncore frequency signal. This confirms that uncore frequency provides an exploitable side channel on these platforms.