PerfCT: Detecting Constant-Time Violations Using Hardware Performance Counters

Clément Médart[1,2], Pierre Graux[1], Clémentine Maurice[2], and Gilles Grimaud[1]

[1] Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
first.last@univ-lille.fr
[2] Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France first.last@inria.fr

Abstract. Side channels constitute a critical threat and are particularly used to attack cryptographic implementations. To prevent these attacks, the cryptographic community has adopted constant-time implementations, which ensure that execution time, control flow, and memory access patterns remain independent of secret values. Developing such implementations is difficult and error-prone, leading to the need for automated constant-time verification techniques. However, state-of-the-art tools suffer from scalability issues.
In this article, we propose PerfCT, a tool that leverages hardware features to detect constant-time violations at the binary level. Since side channels stem from hardware phenomena, we base the detection mechanism directly on features provided by the architecture, namely Hardware Performance Counter (HPCs), a hardware interface that provides high-fidelity observations of various microarchitectural events generated during program execution. Unlike state-of-the-art tools that generate massive, time-dependent execution traces, our methodology collects aggregate, cumulative HPC values only at the end of each program execution. This design yields compact, fixed-size snapshots that characterize the binary's overall microarchitectural footprint independently of its size or execution length. We show that these aggregate HPCs can be used to reliably detect constant-time violations while remaining scalable for implementations of asymmetric algorithms, achieving up to an order of magnitude speedup over state-of-the-art tools. Finally, we found that the optimal set of hardware counters for detection is counter-intuitive, highlighting the subtle complexity of modern architectures where leakages manifest in unexpected microarchitectural components.

Keywords: Constant Time · Side Channels · Vulnerability Detection · Hardware Performance Counters.

1 Introduction

In the cryptographic domain, side channels constitute a powerful source of information leakage [13,18]. Side-channel vulnerabilities pose a significant chal-

lenge to software development. Unlike traditional software vulnerabilities, side-channel vulnerabilities do not arise from incorrect algorithmic behavior. Instead, they stem from the observable effects that computations have on the underlying microarchitecture [15,17,26]. To mitigate these threats, cryptographic software is commonly implemented following the constant-time programming paradigm [4,12]. Constant-time implementations are designed to ensure that control-flow decisions and memory access patterns remain independent of secret data. However, developing such implementations is notoriously difficult, as it requires extensive expertise in side-channel attacks and their underlying leakage mechanisms. As a result, automated verification techniques have become essential for helping developers identify and eliminate constant-time violations.

To detect side-channel vulnerabilities, prior work has proposed two broad classes of approaches: static analysis [2,23] and dynamic analysis [3,6,12,19,24]. Despite significant progress in the literature, existing approaches still face important limitations [8]. For instance, symbolic execution, whether applied to source code or binaries, suffers from scalability issues; when applied to complex programs, such as RSA implementations, it frequently fails to complete execution within a reasonable time budget. Timing-based analyses are inherently limited to timing leakage and cannot detect violations that manifest through other microarchitectural effects, and its execution may run indefinitely until a constant-time violation is observed. Dynamic instrumentation-based approaches introduce additional computation during execution and may perturb the behavior they seek to observe.

Since side-channel leakage ultimately originates from hardware behavior, we argue that constant-time properties should be evaluated directly at the hardware level. Our central hypothesis is that executing a constant-time implementation on hardware must consistently produce the same changes to the microarchitectural state, irrespective of the secret data being manipulated. Consequently, if two distinct executions are distinguishable through differences observed in the microarchitectural state, this reveals a constant-time violation, indicating that the hardware's behavior is dependent on the computed secret data.

Modern processors maintain a rich internal microarchitectural state to support increasingly sophisticated performance optimizations. While this state is not directly observable, commodity processors from Intel and AMD expose Hardware Performance Counters (HPCs), which provide a hardware-supported view of many microarchitectural events, thereby enabling novel hardware-level side-channel analysis. We leverage these counters as a high-fidelity interface to characterize how program executions interact with the underlying hardware. Specifically, our methodology collects aggregated HPC measurements from multiple executions of a target implementation under different secret inputs. By analyzing the changes observed across all distinct executions, we determine whether the implementation exhibits secret-dependent behavior, and therefore whether it is constant-time (CT) or non constant-time (NCT).

Building on this observation, our approach characterizes implementations through their HPC profiles. We repeatedly execute a target implementation us-

ing different secret inputs and collect aggregated HPC measurements for each execution. We then analyze whether executions associated with different secrets remain statistically indistinguishable. Intuitively, a constant-time implementation should produce similar HPC profiles regardless of the processed secret, whereas a non constant-time implementation may leave secret-dependent microarchitectural footprints and therefore a bigger statistical difference. To automate this analysis, we leverage machine learning techniques to distinguish constant-time from non constant-time behaviors and to identify the most informative hardware events for detection. Moreover, this work focuses exclusively on binary analysis, thereby excluding source code analyzers. This choice is motivated by the observation that compiler optimizations can introduce or remove constant-time violations during compilation, making source-level analyses insufficient for detecting compiler-induced leaks [6,7,9,20,21].

We address the following research questions:

- **RQ1**: How can repeated, noisy HPC measurements be used to characterize an implementation's microarchitectural behavior?
- **RQ2**: Which HPCs provide the most informative view of constant-time behavior, and how can they be selected from the large set of available hardware counters?
- **RQ3**: Can HPC-based analysis reliably distinguish constant-time from non-constant-time implementations?

Our primary contributions are summarized as follows:

- **C1: Systematic HPC Feature Selection.** We propose a rigorous method to systematically evaluate and select the most relevant HPCs for constant-time verification. (Section 4.2)
- **C2: Hardware-Centric Framework.** We introduce PerfCT, a hardware-centric framework that leverages machine learning techniques to automate binary-level violation detection. By utilizing cumulative HPC footprints rather than heavy time-dependent instruction tracing, PerfCT captures the aggregate microarchitectural behavior of a program execution in a compact format. PerfCT is available at https://gitlab.univ-lille.fr/2xs/perfct. (Section 4.3)
- **C3: Scalability and State-of-the-Art Evaluation.** We evaluate PerfCT against established dynamic and formal verification tools across a diverse cryptographic benchmark. We demonstrate that our hardware-driven approach scales efficiently to complex asymmetric primitives, achieving up to an order of magnitude speedup and successfully completing the verification where state-of-the-art symbolic execution engines time out. (Section 6)

2 Background

This section introduces the fundamental concepts of hardware side-channel leakages and their exploitation against cryptographic implementations (Section 2.1). We then describe the architectural functionality and operation of HPCs (Section 2.2).

2.1 Side-Channel Attacks

A side-channel attack collects and interprets signals unintentionally emitted by a system to indirectly recover sensitive information that is normally protected or inaccessible. Hardware side-channel attacks specifically target the physical or microarchitectural behavior of components during operation, differentiating them from attacks that exploits bugs or mistake in algorithm logic. A wide variety of side channels exist such as execution time variations [25] and cache timing dependencies [15,17,26].

Wong et al. [25] demonstrated an attack targeting Knuth's [11] modular exponentiation, a routine common in cryptographic implementations. Analyzing this algorithm reveals that an additional arithmetic operation (Figure 1, lines 9-11) is conditionally performed when the secret exponent bit equals 1. Although mathematically trivial, this distinction introduces a significant variation in the algorithm's temporal behavior. As shown by Wong et al., an attacker can accurately measure the time required by the processor to execute these supplementary instructions. Analyzing these temporal variations allows for the deduction of the number of bits set to 1 in the secret exponent. This information leakage constitutes a timing side channel directly correlated with raw execution time.

```
1 uint64_t mod_exp(uint64_t C, // plaintext
2              uint64_t N, // modulus
3              uint64_t d[], // bits of the secret exponent
4              size_t n // number of bits of the exponent
5             ) {
6   uint64_t x = C;
7   for (size_t j = 0; j < n; ++j) {
8     x = (x * x) % N;
9     if (d[j] == 1) {
10      x = (x * C) % N;
11    }
12  }
13  return x;
14 }
```

Fig. 1: Modular Exponentiation Algorithm.

Another critical type of side-channel attacks is cache timing dependency. Due to limited processor cache capacity (L1, L2, L3), the processor frequently performs data evictions. Since L1 and L2 are shared by the same physical core, and L3 is shared across all cores in the CPU package, multiple applications contend for cache space. Techniques such as Prime+Probe [15] or Flush+Reload [26] enable an attacker to distinguish whether a specific data block is present in the cache. This is done by observing the access latency difference between a cache hit

(fast) and a cache miss (slow). Observing these access times permits the indirect inference of memory addresses manipulated by other applications, even if the attacker lacks direct access to the cached data. Consequently, these cache-timing side channels observe modifications to the cache state, potentially leading to the partial or total reconstruction of secret data if there is a dependency between memory accesses and the secret data.

Both categories of these side-channel attacks are critical in advanced security threats. In order to avoid differences when computing distinct input data, and thus preventing side-channels exploitation, the cryptographic community has adopted constant-time implementations. A constant-time implementation ensures that a program's execution behavior is entirely independent of sensitive input data. Specifically, its execution time, control flow, and memory access patterns must remain invariant across distinct runs, thereby eliminating microarchitectural variations that could leak cryptographic secrets.

2.2 Performance Monitoring Counters

The increasing complexity of modern processors makes precise performance measurement challenging for software developers. To address this, Intel and AMD incorporate a Performance Monitoring Unit (PMU) [22]. While Hardware Performance Counters (HPCs) and Performance Monitoring Counters (PMCs) are used synonymously in the literature, terminology varies across architectures. We adopt the term PMC, following Intel and AMD processors documentation. The PMU facilitates the monitoring of hardware performance events via PMCs. PMCs are hardware counters that record the occurrence of specific microarchitectural events, such as cache misses, branch mispredictions, or memory access conflicts. These counters are widely used by performance profiling tools, such as Intel VTune Profiler, for event-based sampling and hardware resource utilization analysis.

A PMC is defined as a single numerical value representing the total occurrence count of a specific microarchitectural event during a monitored execution. Operationally, the PMU increments the corresponding counter by one each time the specific event is observed. To configure the monitoring, events are selected via MSRs (e.g., `IA32_PERFEVTSELx` MSRs on Intel). The collected results are stored in other MSRs (e.g., `IA32_PMCx` MSRs on Intel). These counter registers can be read (result retrieval) and written (setting predefined values or resetting the count). The PMU supports observation flexibility by allowing the activation or deactivation of monitoring at specified time points. Modern processors typically allow to monitor simultaneously between five and ten PMCs.

3 Related Work

This section outlines related work on automated constant-time verification tools (Section 3.1) as well as the use of PMCs as a hardware interface for side-channel attack detection (Section 3.2).

3.1 Binary Analysis for Constant-Time Properties

While constant-time verification can be performed at the source level, analyzing the compiled binary is essential to detect leakages introduced by the compiler [6,7,9,20,21] or to evaluate closed-source third-party libraries. Existing binary-level approaches generally fall into two categories: formal verification and dynamic analysis. Despite significant advancements, both face severe scalability limitations when handling complex cryptographic implementations. Below, we review a representative subset of these solutions, focusing on five tools evaluated in recent benchmarking literature [8] that cover the full spectrum of binary-level constant-time analysis while highlighting their respective constraints.

Binsec/Rel [6] is presented as the first automated, efficient, and correct tool for constant-time analysis at the binary level. It employs relational symbolic execution, combined with specific optimizations tailored for binary analysis. Instead of analyzing a single execution path, Binsec/Rel models two execution paths that follow the same symbolic trace. The objective is to prove that the leakage of both traces remains identical, which guarantees the CT property of the implementation. However, as a symbolic execution engine, Binsec/Rel suffers from path explosion and solver bottlenecks when scaling to large binaries. Consequently, it fails to analyze complex asymmetric cryptographic implementations (e.g., Mbed TLS RSA) within a reasonable time budget.

Abacus [3] quantifies information leakage based on the reduction of attacker uncertainty, measuring the number of possible secret key options eliminated by side-channel observations. The analysis proceeds in three steps. First, an execution trace is obtained using the binary instrumentation framework Intel Pin [14]. Second, a specialized symbolic execution engine translates leak signals (i.e., secret-dependent control flows or data accesses) into symbolic constraints; only instructions manipulating secret values are analyzed symbolically, while others use concrete values. Finally, Abacus estimates the leaked bit count. Like Binsec/Rel, Abacus faces significant scalability challenges due to symbolic execution.

Ctgrind [12] is built upon Valgrind, a memory error detection tool. Valgrind can track uninitialized memory variables by highlighting their usage in control flow and memory access. Ctgrind leverages this mechanism by marking the secret as uninitialized, thereby enabling Valgrind to report usage of secrets in the program's control flow and memory accesses. While Ctgrind scales efficiently, requiring less than seconds to execute large asymmetric cryptographic implementations, its strict taint-tracking logic suffers from a high false-positive rate. Thus, it frequently flags benign operations that do not constitute actual microarchitectural leaks, severely complicating manual code auditing [8].

MicroWalk [24] relies on dynamic binary instrumentation using Intel Pin [14]. To detect leaks, it logs execution traces and computes the Mutual Information (MI) [10] between the secret input and the observed traces. A non-zero MI score reveals a statistical dependence, signaling a potential information leakage. However, this approach faces significant scalability bottlenecks: the overhead of instruction-level instrumentation combined with the storage and processing

of massive execution traces makes MicroWalk substantially slower than taint-analysis tools like Ctgrind.

DudeCT [19] is a compact utility (approximately 300 lines of C) designed to assess the temporal variability of code execution on a given platform. DudeCT detects constant-time violations by repeatedly measuring the execution time for two different classes of input data. It applies a statistical test, indicating a non-constant-time execution violation if the comparison of distribution results shows a significant difference. While DudeCT imposes no architectural restrictions on program size, its practical scalability is bottlenecked by statistical convergence. Lacking a definitive stopping criterion, the analysis effectively runs indefinitely when no violation is present. Furthermore, its detection capabilities are strictly limited to global execution time variations, leaving it blind to microarchitectural leaks that do not visibly alter the total runtime.

3.2 Hardware-Based Side-Channel Detection using PMCs

To the best of our knowledge, no previous study has leveraged PMCs to verify the constant-time property of an implementation. Existing PMC-based frameworks in the literature focus exclusively on detecting active, ongoing side-channel attacks at the system level [5,16], rather than verifying the correctness of a single program. These defense mechanisms typically monitor low-level microarchitectural events in real time and employ machine learning to distinguish a malicious attacker's footprint from legitimate execution. While these tools act as runtime detection and mitigation systems against external threats, our work serves as a pre-deployment verification utility to guarantee that the application itself does not inherently leak secrets.

4 Methodology

In this work, we propose a hardware-centric methodology to characterize and verify the constant-time properties of cryptographic implementations. Our approach leverages PMCs to detect CT violations by observing changes in the implementation's execution behavior when processing different secret inputs.

Our methodology offers two primary advantages. First, since PMCs operate directly at the processor level, we can capture the execution snapshot of any arbitrary binary. This makes our approach applicable to numerous widely used commercial applications where the source code is restricted. Second, the snapshots collected from PMCs are easily analyzed because they are represented by single numerical values contrary to numerous state of the art approaches that handle huge traces.

Our working hypothesis is that CT and NCT implementations induce different statistical signatures in PMC measurements when they are executed with identical or distinct secret inputs. More precisely, we assume that the way PMC measurements are distributed across repeated executions, and the way these

distributions change when secret inputs are kept identical or varied, provide a characteristic profile of the implementation.

Let C be an implementation and let I denote a secret input value drawn from an input domain D. For a given set of PMCs, we denote by

$$\mathrm{PMC}(C, I)$$

the vector of aggregate counter values measured when executing C with secret input I. Since PMC measurements are noisy, partial, and architecture-dependent observations of processor activity, a single execution is not sufficient to characterize the behavior of C. Instead, for each secret input, we collect repeated measurements and consider the resulting empirical distributions.

Given two secret inputs I_a and I_b, we then compare the distributions obtained from repeated executions of C with these inputs. These comparisons are summarized through statistical descriptors, denoted abstractly as

$$\Delta_{\mathrm{PMC}}(C, I_a, I_b),$$

which capture how the observed PMC distributions differ when executions are performed with identical or distinct secret values. The central assumption of our approach is that the statistical profile formed by these descriptors is different for CT and NCT implementations.

This hypothesis is motivated by the fact that CT implementations are subject to unusually strong design constraints. In order to avoid secret-dependent control flow and secret-dependent memory accesses, they often rely on specific programming idioms, regular execution patterns, and transformations that differ from those used in conventional software implementations, including NCT cryptographic implementations. As a result, CT code is expected to induce distinctive microarchitectural effects during execution. These effects are not characterized by a single PMC value or by one isolated event, but by the statistical structure of the PMC measurements observed over repeated executions. By comparing executions with identical and distinct secrets, we aim to reveal this structure and use it as a discriminating signature between CT and NCT implementations.

It is important to stress what this hypothesis does not imply. We assume that PMC measurements can capture effects related to the secret inputs, even when the implementation follows CT programming constraints. However, we do not claim that a CT implementation always produces the same PMC values for different secret inputs. More importantly, we do not even assume that PMC measurements vary less for CT implementations than for NCT implementations when the secret changes. The central assumption is only that they vary differently: the statistical profile of PMC variations, observed over repeated executions and across pairs of identical or distinct secrets, is expected to be significantly different when the implementation is CT compared to when it is NCT.

In this section, we describe a methodology that uses PMC measurements to distinguish CT implementations from NCT ones through these statistical profiles. Section 4.1 describes how PMC measurements are collected and processed in order to obtain a compact characterization of an implementation's behavior

for selected secret inputs. Section 4.2 details how to select the subset of PMCs that provides the most informative characterization. Finally, Section 4.3 explains how these characterizations are used to automate CT violation detection.

4.1 Theoretical Measurement and Data Processing

To characterize a cryptographic implementation's behavior, we first quantify the occurrence of specific microarchitectural events during execution. To perform this quantification, we retrieve the value of given PMCs before and after executing the tested implementation. The difference between these two values is the number of microarchitectural events generated during the run.

However, given the inherent complexity of modern CPUs and multi-threaded environments, the raw numerical values returned by PMCs are subject to measurement noise. To mitigate this noise, we execute the implementation N_{run} times using the same fixed secret input. For each secret input, the N_{run} individual event occurrence counts are aggregated into a set. This set defines an empirical distribution that represents the typical behavior of the implementation for that specific secret input.

Since we aim to characterize behavioral differences between empirical distributions obtained using distinct secret inputs, we collect multiple behavior measurements (N_{run}) for each selected secret value (N_{key}). Repeated executions with the same secret input provide the within-secret measurement distribution, which captures the variability due to measurement noise and secret-independent effects. Comparisons between empirical distributions obtained from distinct secret inputs then capture how this variability changes when the secret is modified. To convert this collection ($N_{run} \times N_{key}$) of behavior measurements into compact, quantifiable metrics, we derive change metrics for every pair of distinct secret inputs ($(N_{key} \times (N_{key} - 1))/2$). This conversion focuses on a small number of statistical metrics that summarize both changes in central tendency and changes in the overall shape of the distributions, rather than studying all individual measurement values. This representation reduces the amount of data manipulated by the subsequent classification steps: instead of using all raw PMC measurements, we retain only a small number of statistical descriptors for each pair of selected secrets. This approach significantly reduces the computational complexity because the number of secrets needed to be tested is in practice lower than the required number of measurements ($N_{run} = 200$, $N_{key} = 16$).

The change metrics are represented by a pair of statistical features: the absolute mean difference and the Kolmogorov-Smirnov (KS) test. The absolute mean difference provides a direct measure of the change in central tendency between the two distributions. We employed the KS test because we also want to take into account the shape of the distribution of PMC measurements. This method outlines the procedure for measuring a single microarchitectural event. To characterize the implementation's behavior across various events, we repeat this procedure for each available event. Consequently, for each event, a distinct change metric vector is obtained from the implementation measurement. The

collection of these change metrics over all selected PMCs forms the statistical profile used in the subsequent PMC selection and CT/NCT classification steps.

4.2 Performance Monitoring Counters Selection

The PMU records a vast number of microarchitectural events. To efficiently identify the PMCs that best characterize the statistical differences between CT and NCT implementations, we employ a multi-stage feature prioritization technique to manage computational complexity and the high dimensionality inherent in evaluating all possible combinations.

First, we collect measurement data for all available microarchitectural events from different cryptographic implementations known to be either CT or NCT. We then apply our vectorization methodology to these measurements, generating a comprehensive set of change metric feature vectors. These vectors, paired with their corresponding ground truth labels (CT or NCT), are used to train a Random Forest classifier. The Random Forest classifier identifies the most influential PMCs, i.e., those exhibiting the highest feature importance in distinguishing CT from NCT statistical profiles, thereby reducing the initial feature space dimensionality.

Following the feature ranking step, we systematically evaluate combinations of the high-importance PMCs. This combinatorial evaluation employs the Linear SVM classifier method detailed in Section 4.3. Specifically, we use the Linear SVM to determine the set of PMCs that maximizes classification performance. The final PMC set is selected based on the highest balanced accuracy score achieved by the Linear SVM model, ensuring that the chosen PMCs provide the most discriminative information for distinguishing CT and NCT implementations.

4.3 Automated Constant-Time Violation Detection

To perform constant-time violation detection, we propose a method that distinguishes statistical profiles associated with CT implementations from those associated with NCT implementations. Rather than relying on a single PMC value or on the magnitude of a single variation, the method exploits the change metric vectors introduced above as signatures of the implementation behavior.

Based on our hypothesis, we expect to observe distinct statistical structures when analyzing the change metrics across empirical distributions obtained from known CT and NCT implementations. These structures are not assumed to correspond to low variation for CT implementations and high variation for NCT implementations. Instead, they are expected to reflect different patterns of PMC variation when executions are compared across identical and distinct secret inputs. To solve this binary classification problem, we employ a Linear Support Vector Machine (Linear SVM). The use of Linear SVM is motivated by its computational efficiency and its suitability for separating CT and NCT statistical profiles in the selected feature space.

Before the training phase, a dataset representing implementation behaviors must be gathered. For each implementation, secret input, and selected microarchitectural event, we generate the corresponding empirical distribution of PMC measurements. The classification configuration utilizes a cross-validation approach: all change metrics from other implementations are used for training, while the change metrics from the implementation currently being classified are reserved for testing. This cross-validation approach evaluates the ability of our detection method to generalize to implementations not seen during training. After training, we evaluate the performance of the CT/NCT prediction by calculating the balanced accuracy of the SVM predictions. Classification performance is therefore measured using balanced accuracy across all cross-validation folds.

5 Implementation

The PerfCT framework architecture is divided into two principal components, a low-level kernel module and a high-level Python orchestration script. The kernel module is responsible for configuration and management of the Model-Specific Registers (MSRs), since direct access to MSRs is restricted to kernel space. Moreover, the kernel module implements sophisticated event tracing logic to ensure data integrity. Indeed, it monitors the target application's scheduling events using kernel tracepoint events. When the target application is scheduled out, the kernel module halts and saves the active PMC configuration; conversely, upon scheduled in, it restores and enables the previously saved PMC state. This mechanism ensures the precise capture of microarchitectural events exclusively generated by the target process. The Python script acts as the experiment manager, orchestrating the execution flow. It manages the experimental behavior by driving the execution collection of various targets with predefined inputs. The target application (e.g., cryptographic implementations) is executed via a wrapper that initiates the cryptographic process and extracts the relevant PMC values collected by the kernel module.

To significantly minimize the noise and maximize the isolation of the tested implementation's microarchitectural footprint, we employ several Linux system controls. We leverage Systemd and Cgroups, coupled with the `AllowedCPUs` setting, to dedicate a single processor core exclusively to the target application. This core affinity is enforced using the kernel API `set_cpus_allowed_ptr`. Furthermore, we utilize the `nohz_full` parameter within the Linux command line to minimize unnecessary scheduling events. These combined measures drastically reduce external noise from other background processes that contaminating the monitored microarchitectural state. Finally, to reduce noise due to kernel interruptions, the PMCs are configured to record microarchitectural events only when the processor is in user space mode (e.g., Ring 3).

The open-source code and measurement dataset are available at https://gitlab.univ-lille.fr/2xs/perfct.

6 Evaluation

To evaluate the effectiveness and practicality of our proposed methodology, we address the following research questions in subsequent sections:

- **RQ1** (Section 6.1): How can repeated, noisy PMC measurements be used to characterize an implementation's microarchitectural behavior?
- **RQ2** (Section 6.3): Which PMCs provide the most informative view of constant-time behavior, and how can they be selected from the large set of available hardware counters?
- **RQ3** (Section 6.2): Can PMC-based analysis reliably distinguish constant-time from non-constant-time implementations?

Benchmark. To answer these questions, we have employed *Geimer et al.*'s unified benchmark dataset [8] comprising 18 different cryptographic implementations. From this dataset, we extracted a balanced dataset that features an equal number of CT and NCT implementations and includes both symmetric and asymmetric cryptographic algorithms. To maintain comparability with state-of-the-art tools, we reused the set of secret inputs from *Geimer et al.* (e.g., 16 secret keys) for the measurements.

Setup. All measurements were conducted on a Honor HLYL-WFQ9 device, with an AMD Ryzen 5 4600H processor. To ensure reliable data acquisition and consistent microarchitectural behavior, the processor boost was disabled, and the CPU frequency was fixed at 1.4 GHz.

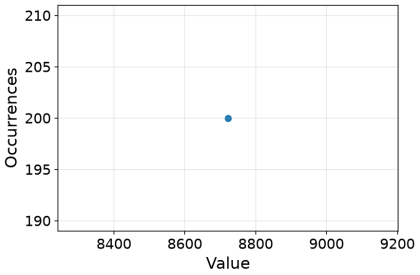
6.1 PMC Characterization

In this section, we address **RQ1** by characterizing the variability of repeated PMC measurements and by showing why PMC values must be handled as empirical distributions rather than as single deterministic values.

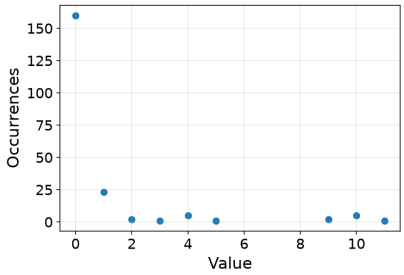
To characterize the noise behavior of PMCs, we repeatedly execute a single implementation with a fixed secret input and collect 200 measurements for each selected PMC. Figure 2 shows the resulting distribution of PMC values for three distinct PMCs. The y-axis represents the absolute frequency of a specific count value. The x-axis displays the value for the PMC. Each data point on the graph corresponds to the frequency of a single count value. Our analysis shows that PMCs naturally fall into three categories according to their susceptibility to measurement noise.

The first category, illustrated in Figure 2a, consists of highly stable PMCs. These counters exhibit minimal sensitivity to ambient measurement noise and can potentially be used directly without post-processing stage due to their high data fidelity.

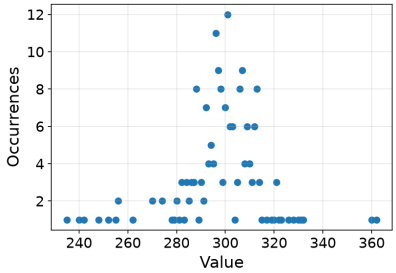
The second category, illustrated in Figure 2b, consists of PMCs exhibiting low levels of measurement noise. The vast majority of the recorded values are identical. We observe minor spikes above this majority. This slight increase in



(a) Bearssl ecdsa: Retired Near Returns



(b) Bearssl poly chacha (SSE2): Data Cache Refills from System



(c) mbedtls3 aes cbc (T): Dynamic Tokens Dispatch Stall Cycles 0

Fig. 2: Characterization of the noise behavior of PMCs.

values can typically be attributed to the multi-threaded nature of modern processors. Specifically, these variations may result from shared hardware resources being utilized by background processes, which introduces minor contention and slightly slows down the execution of the measured implementation.

The third category, illustrated in Figure 2c, consists of PMCs that are significantly affected by ambient noise. Data collected from these counters requires rigorous post-processing techniques, such as smoothing or filtering, before reliable use in constant-time detection.

6.2 Selection of Performance Monitoring Counters

In this section, we address **RQ2** by identifying the most relevant set of PMCs for detecting constant-time behaviors.

We restrict the selected PMC set to five events. This choice reflects a common hardware constraint: modern processors expose only a limited number of programmable PMCs that can be monitored simultaneously. For instance, the AMD Ryzen 5 4600H processor used in our experiments supports monitoring five programmable events concurrently (the sixth PMC register is reserved by the Linux scheduler). Restricting the analysis to five PMCs therefore keeps our methodology compatible with typical hardware capabilities while ensuring that

- `0x47`: Misaligned loads
- `0x43`: Data Cache Refills from System
- `0x5a`: Hardware Prefetch Data Cache Fills
- `0x40`: Data Cache Accesses
- `0xaf`: Dynamic Tokens Dispatch Stall Cycles 0

	pred NCT	pred CT
true NCT	88.33%	11.67%
true CT	2.43%	97.57%

Fig. 3: PMCs Selected for Symmetric Algorithms and Corresponding Accuracy

- `0xd1`: Retired Conditional Branch Instructions
- `0xc4`: Retired Taken Branch Instructions
- `0x35`: Store to Load Forward
- `0xc8`: Retired Near Returns
- `0x24`: Bad Status 2

	pred NCT	pred CT
true NCT	100.00%	0.00%
true CT	0.00%	100.00%

Fig. 4: PMCs Selected for Asymmetric Algorithms and Corresponding Accuracy

all measurements can be collected in a single execution campaign with minimal overhead.

We derive separate PMC sets for symmetric and asymmetric cryptographic implementations. This distinction is motivated by the fact that these two classes of algorithms stress different microarchitectural resources. Symmetric cryptography primarily relies on operations such as modular arithmetic, substitutions, and permutations, whereas asymmetric cryptography performs large-integer or elliptic-curve arithmetic involving different execution units and memory access patterns. Consequently, the PMCs that best characterize constant-time behavior are not necessarily the same for both classes of algorithms.

To identify the most informative PMCs for each class of cryptographic implementations, we perform the following selection procedure separately for symmetric and asymmetric algorithms. For each cryptographic implementation and each selected secret key, we collect 200 execution measurements across all 61 available PMCs. The choice of 200 executions balances measurement cost against the need to capture measurement variability. We then apply the methodology detailed in Section 4.2: a Random Forest model first ranks PMCs according to their importance, after which a Linear SVM identifies the subset that maximizes balanced accuracy. This process yields two sets of PMCs, one for symmetric cryptography (Figure 3) and one for asymmetric cryptography (Figure 4).

Figures 3 and 4 present the confusion matrices obtained with the selected PMC sets for symmetric and asymmetric cryptographic implementations, respectively. For symmetric algorithms, the selected PMC set achieved a high precision of 97.57% when classifying constant-time behavior. However, the 11.67% error rate when misclassifying non-constant-time implementations can be attributed to the fact that the information leakage is too small to allow Linear SVM to separate the corresponding statistical profiles. For asymmetric algorithms, the selected PMC set achieved perfect classification accuracy. This result is likely due both to the smaller number of implementations, which simplifies the identifica-

Table 1: Official descriptions of the most discriminative PMCs provided by AMD.

PMC	AMD description
`0xaf`	Cycles where a dispatch group is valid but does not get dispatched due to a token stall.
`0x47`	Misaligned loads.
`0xc4`	The number of taken branches that were retired. This includes all types of architectural control flow changes, including exceptions and interrupts.
`0x35`	Number of STLF hits.
`0xc8`	The number of near return instructions (RET or RET Iw) retired.
`0x24`	Store To Load Interlock (STLI) are loads that were unable to complete because of a possible match with an older store, and the older store could not do STLF for some reason. There are several reasons why this occurs, and this perfmon organizes them into three major groups.

tion of an effective PMC set, and to the greater homogeneity of the asymmetric algorithms in our dataset. Note that the confusion matrices in Figures 3 and 4 represent the accuracy of the PMC set selected using the entire set of implementations. These results do not reflect PerfCT's accuracy when applied to unseen implementations.

Our evaluation demonstrates that the most effective PMCs for detecting constant-time violations are often counter-intuitive. For instance, PMCs that would appear naturally suited to capturing control-flow leakage, such as `0x076` (counting cycles not in halt) or `0x082` (counting 64-byte instruction cache lines fulfilled from L2), were not among the most informative counters. For symmetric algorithms, the PMCs `0xaf` (Dynamic Tokens Dispatch Stall Cycles 0) and `0x47` (Misaligned loads) emerged as the most discriminative counters, whereas for asymmetric algorithms, PMCs `0xd1` (Retired Conditional Branch Instructions), `0xc4` (Retired Taken Branch Instructions), and `0x24` (Bad Status 2) proved more relevant. A similar trend is observed for memory-related events: while the selected PMCs for symmetric algorithms, i.e., `0x43` (Data Cache Refills from System), `0x5a` (Hardware Prefetch Data Cache Fills), and `0x40` (Data Cache Accesses), align with expectations, asymmetric algorithms are better characterized by less obvious counters such as `0x24` (Bad Status 2) and `0x35` (Store to Load Forward).

Explaining why these seemingly unrelated PMCs are highly discriminative remains challenging. The AMD manual for the Ryzen 5 4600H CPU provides only brief descriptions of these events, preventing a detailed hardware-level interpretation of the observed correlations. For completeness, we report the official descriptions in Table 1.

Table 2: Vulnerability detection. S: status (● secure; ○ insecure; ◐ unknown), 💣: the tool crashed, ⌛: the analysis timed-out after 3600 s, C: confidence index. Numbers from Binsec/Rel, Abacus, ctgrind, Microwalk and dudect are from [8].

	Binsec	Abacus	ctgrind	MicroWalk	dudect	PerfCT		Ground Truth
Benchmarks	S	S	S	S	S	C	S	
bearssl aes cbc (T)	○	○	○	○	○	0.93	○	○
bearssl aes gcm (T)	○	○	○	○	○	0.56	○	○
bearssl aes cbc (BS)	●	●	◐	◐	◐ (⌛)	0.62	○	●
bearssl aes gcm (BS)	●	●	◐	◐	◐ (⌛)	0.98	●	●
bearssl poly chacha (CT)	●	◐	◐	◐	◐ (⌛)	0.99	●	●
bearssl poly chacha (SSE2)	●	◐	◐	◐	◐ (⌛)	1.00	●	●
mbedtls3 aes cbc (T)	○	○	○	○	○	0.97	○	○
mbedtls3 poly chacha	●	◐	◐	◐	◐ (⌛)	0.99	●	●
mbedtls3 aes gcm (T)	○	○	○	○	○	0.87	○	○
openssl3 aes cbc (T)	○	○	○	○	◐ (⌛)	0.61	○	○
openssl3 chacha20	●	◐	◐	◐	◐ (⌛)	1.00	●	●
openssl3 poly1305	●	◐ (💣)	◐	◐	◐ (⌛)	0.93	●	●
openssl3 aes cbc (VP)	●	◐	◐	◐	◐ (⌛)	1.00	●	●
bearssl ecdsa	○ (⌛)	◐	○	◐	◐ (⌛)	1.00	●	●
mbedtls3 rsa (oaep)	○ (⌛)	◐ (💣)	○	○	◐ (⌛)	0.97	○	○
mbedtls3 rsa (pkcs1)	○ (⌛)	◐ (💣)	○	○	◐ (⌛)	0.97	○	○
openssl3 ecdsa	○ (⌛)	◐	○	○	◐ (⌛)	0.93	○	●
openssl3 ed25519	●	◐	◐	○	◐ (⌛)	1.00	●	●

6.3 Evaluating Constant-Time Detection

In this section, we address **RQ3** by evaluating the accuracy of our method in differentiating CT implementations from NCT implementations.

We evaluate the effectiveness of PerfCT in identifying constant-time (CT) violations using the methodology detailed in Section 4.3. Using the tailored PMC subsets selected in Section 6.2, we classify each target implementation by processing 200 execution measurements per selected secret key according to our vectorization and SVM classification pipeline. We benchmark PerfCT against five state-of-the-art binary-level verification tools–Binsec, Abacus, Ctgrind, MicroWalk, and Dudect–using the standardized dataset compiled by Geimer et al. [8], including our necessary adaptations presented at the beginning of the section. For the baseline tools, we inherit the evaluation outcomes reported in [8]

Following Geimer et al. [8], we classify an implementation as secure (●) if no existing tool reports a relevant constant-time violation; otherwise, it is classified as insecure (○). We exclude false positives reported by tools that do not indicate a true constant-time violation. We do not evaluate real-world exploitability, i.e., an implementation may be marked as insecure based on a constant-time violation, even if the information leak is not considered exploitable in practice.

We introduce a Confidence Index ($C \in [0, 1]$) representing the frequency of the SVM's majority vote over repeated classification trials (e.g., $C = 0.80$ means the predicted label was assigned in 80% of the runs).

PerfCT achieves a high alignment with the established ground truth, successfully identifying standard constant-time violations across both symmetric and asymmetric primitives (Table 2). This suggests that the combination of

specific PMCs and the Linear SVM model is highly effective in discriminating between constant-time and non-constant-time behavior. Specifically, PerfCT correctly flags all table-lookup implementations (labeled 'T') as insecure, including AES-CBC and AES-GCM across BearSSL and mbedTLS, and AES-CBC on OpenSSL. More importantly, for asymmetric schemes such as MbedTLS RSA and OpenSSL ed25519 implementations, PerfCT's verdicts perfectly match the ground truth. This demonstrates that our hardware-centric approach scales efficiently to complex, large-integer arithmetic where software-based techniques like relational symbolic execution frequently time out.

Despite its high accuracy, PerfCT introduces false positives on specific implementations due to benign microarchitectural variations and algorithmic masking techniques (Table 2). First, PerfCT misclassifies BearSSL AES-CBC (BS) ($C = 0.62$), which suggests that while the selected PMCs are effective at capturing behavioral changes, they may also be sensitive to specific implementation artifacts or benign microarchitectural noise that is not strictly indicative of a timing leak exploitable in practice. Second, PerfCT flags OpenSSL ECDSA as insecure ($C = 0.93$), a discrepancy shared by state-of-the-art tools including Binsec/Rel, Ctgrind, and MicroWalk. This false positive stems from OpenSSL's use of cryptographic blinding, which inherently introduces operand-dependent execution variations on randomized values. While these variations trigger constant-time detectors, the underlying leak is structurally unexploitable. This highlights a pervasive limitation in automated verification: our approach, like other state-of-the-art tools, successfully detects the behavioral non-determinism, but cannot inherently deduce the mathematical security provided by blinding mechanisms.

6.4 Runtime Evaluation

PerfCT achieves execution times compatible with real-world Continuous Integration and Continuous Deployment (CI/CD) pipelines, successfully decoupling verification overhead from binary complexity (Table 3). Our performance evaluation distinguishes between two phases: raw measurement collection time (M) and downstream analysis time (A). Although our statistical methodology requires executing each target multiple times to filter microarchitectural noise, the total overhead remains low enough for practical developer workflows.

Crucially, while data collection time scales naturally with the target's native execution runtime, PerfCT guarantees a constant analysis time ($O(1)$). The analysis process itself always takes between two and four seconds because the input matrix for the Linear SVM has a fixed size, independent of the binary's complexity. For symmetric ciphers, PerfCT requires approximately five seconds; this is slightly slower than several state-of-the-art tools due to our noise-reduction execution loops, but the overhead remains negligible. For complex asymmetric algorithms, the advantage becomes decisive: native execution under PMC monitoring completely bypasses the instrumentation overhead of dynamic binary translation frameworks like MicroWalk (with Intel Pin). More importantly, PerfCT fundamentally eliminates the path explosion problem that plagues symbolic execution

Table 3: Vulnerability detection performance. M: measurement collection time, A: analysis time. ⧗: the analysis timed-out after 3600 s. Numbers from Binsec/Rel, Abacus, ctgrind, Microwalk and dudect are from [8].

	Time (s)						
Benchmarks	Binsec	Abacus	ctgrind	MicroWalk	dudect	PerfCT	
						M	A
bearssl aes cbc (T)	0.10	0.65	0.16	1.39	100.51	2.04	3.76
bearssl aes gcm (T)	0.22	0.22	0.18	1.51	8.69	2.09	3.78
bearssl aes cbc (BS)	0.31	0.31	0.17	1.55	⧗	2.21	3.76
bearssl aes gcm (BS)	0.48	0.48	0.17	1.73	⧗	2.27	3.77
bearssl poly chacha (CT)	0.18	1.3	0.16	1.43	⧗	2.07	3.76
bearssl poly chacha (SSE2)	0.11	1.17	0.15	1.34	⧗	2.09	3.79
mbedtls3 aes cbc (T)	0.17	0.17	0.19	1.54	0.63	2.06	3.73
mbedtls3 poly chacha	0.49	5.02	0.18	2.95	⧗	2.75	3.78
mbedtls3 aes gcm (T)	0.4	0.4	0.19	1.90	0.89	2.21	3.75
openssl3 aes cbc (T)	0.16	3.02	0.18	1.38	⧗	2.03	3.80
openssl3 chacha20	0.09	3.77	0.15	1.36	⧗	2.01	3.73
openssl3 poly1305	0.35	11.19	0.17	1.70	⧗	2.24	3.70
openssl3 aes cbc (VP)	0.59	1.01	0.19	1.68	⧗	2.24	3.77
bearssl ecdsa	⧗	246.62	0.41	109.31	⧗	49.96	2.54
mbedtls3 rsa (oaep)	⧗	2203.89	0.98	847.27	⧗	249.32	2.60
mbedtls3 rsa (pkcs1)	⧗	2193.2	0.96	826.23	⧗	258.33	2.61
openssl3 ecdsa	⧗	202.72	1.22	13.76	⧗	6.34	2.56
openssl3 ed25519	25.63	59.87	1.03	9.86	⧗	6.06	2.64

methods such as Binsec/Rel and Abacus. For instance, on MbedTLS RSA, Binsec/Rel times out after 3600 seconds, and Abacus requires over 2200 seconds, whereas PerfCT delivers a verdict in less than 261 seconds, achieving a speedup of up to one order of magnitude.

7 Future Work

One limitation of this work is that our evaluation is restricted to the AMD Ryzen 5 4600H. Although the methodology is processor-independent, evaluating PerfCT on ARM and RISC-V architectures would provide stronger evidence of its architecture-agnosticism and clarify which aspects of the framework require processor-specific adaptation.

Furthermore, Intel Core Ultra Processors Series 2 represent a compelling evaluation target; their significantly larger event set (309 vs. 61 [1]) and expanded simultaneous monitoring capability may enable richer microarchitectural characterizations and more efficient data collection compared to the Ryzen 5 4600H.

8 Conclusion

In our paper we presented PerfCT, a hardware-centric framework for detecting constant-time violations using PMCs. Rather than relying on instruction tracing

or symbolic reasoning, PerfCT characterizes program executions through their aggregate microarchitectural footprints, enabling efficient binary-level verification directly on commodity processors.

To make this approach practical, we introduced a systematic methodology for selecting the PMCs that best capture constant-time behavior. Our analysis revealed that the most informative counters are not necessarily those intuitively expected; thus, PerfCT uses Linear SVM techniques to automatically identify the discriminative PMCs and classify implementations as constant-time or non-constant-time. Our evaluation demonstrates that PerfCT remains effective even when precise methods encounter timeouts, achieving up to an order-of-magnitude reduction in analysis time for complex asymmetric cryptographic implementations where symbolic execution is prohibitively expensive. Thus, PerfCT can serve as a preliminary tool to guide the application of time-intensive analysis tools toward problematic code.

Overall, our results show that PMCs constitute a practical foundation for hardware-level constant-time verification, particularly in CI/CD environments. We hope this work encourages further exploration of hardware-assisted security, bridging the gap between software verification techniques and the microarchitectural phenomena from which side-channel vulnerabilities arise.

Disclosure of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Intel core processors, arrow lake hybrid events, core events. https://perfmon-events.intel.com/platforms/arrowlake/core-events/p-core/
2. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX Security Symposium. pp. 53–70. USENIX Association (2016)
3. Bao, Q., Wang, Z., Li, X., Larus, J.R., Wu, D.: Abacus: Precise side-channel analysis. In: ICSE. pp. 797–809. IEEE (2021)
4. Barthe, G., Betarte, G., Campo, J.D., Luna, C.D., Pichardie, D.: System-level non-interference for constant-time cryptography. In: CCS. pp. 1267–1279. ACM (2014)
5. Carnà, S., Ferracci, S., Quaglia, F., Pellegrini, A.: Fight hardware with hardware: Systemwide detection and mitigation of side-channel attacks using performance counters. DTRAP **4**(1), 5:1–5:24 (2023)
6. Daniel, L., Bardin, S., Rezk, T.: Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In: SP. pp. 1021–1038. IEEE (2020)
7. Geimer, A., Maurice, C.: Fun with flags: How compilers break and fix constant-time code. CoRR **abs/2507.06112** (2025)
8. Geimer, A., Vergnolle, M., Recoules, F., Daniel, L., Bardin, S., Maurice, C.: A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries pp. 1690–1704 (2023)

9. Gerlach, L., Pietsch, R., Schwarz, M.: Do compilers break constant-time guarantees? In: FC (2). Lecture Notes in Computer Science, vol. 15752, pp. 327–344. Springer (2025)
10. Guiasu, S.: Information theory with new applications.
11. KnuthD, E.: The art of computer programming (volume 2) (1981)
12. Langley, A.: ctgrind checking that functions are constant time with valgrind, 2010. URL https://github.com/agl/ctgrind **84** (2010)
13. Lou, X., Zhang, T., Jiang, J., Zhang, Y.: A survey of microarchitectural side-channel vulnerabilities, attacks and defenses in cryptography. CoRR **abs/2103.14244** (2021)
14. Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI. pp. 190–200. ACM (2005)
15. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: CT-RSA. pp. 1–20. Lecture Notes in Computer Science, Springer (2006)
16. Payer, M.: Hexpads: A platform to detect "stealth" attacks. In: ESSoS. pp. 138–154. Lecture Notes in Computer Science, Springer (2016)
17. Percival, C.: Cache missing for fun and profit (2005)
18. Pulkit, A., Naval, S., Laxmi, V.: A survey of side-channel attacks in context of cache - taxonomies, analysis and mitigation. CoRR **abs/2312.11094** (2023)
19. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: DATE. pp. 1697–1702. IEEE (2017)
20. Schneider, M., Lain, D., Puddu, I., Dutly, N., Capkun, S.: Breaking bad: How compilers break constant-time implementations. In: AsiaCCS. pp. 1690–1706. ACM (2025)
21. Simon, L., Chisnall, D., Anderson, R.J.: What you get is what you C: controlling side effects in mainstream C compilers. In: EuroS&P. pp. 1–15. IEEE (2018)
22. Sprunt, B.: The basics of performance-monitoring hardware. IEEE Micro **22**(4), 64–71 (2002)
23. Sung, C., Paulsen, B., Wang, C.: CANAL: a cache timing analysis framework via LLVM transformation. In: ASE. pp. 904–907. ACM (2018)
24. Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B.: Microwalk: A framework for finding side channels in binaries. In: ACSAC. pp. 161–173. ACM (2018)
25. Wong, W.H.: Timing attacks on RSA: revealing your secrets through the fourth dimension. ACM Crossroads **11**(3), 5 (2005)
26. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security Symposium. pp. 719–732. USENIX Association (2014)