Investigating Black-Box Function Recognition Using Hardware Performance Counters

Carlton Shepherd, Benjamin Semal, and Konstantinos Markantonakis

Abstract—This paper presents new methods and results for recognising black-box program functions using hardware performance counters (HPC), where an investigator can invoke and measure function calls. Important use cases include analysing compiled libraries, e.g. static and dynamic link libraries, and trusted execution environment (TEE) applications. We develop a generic approach to classify a comprehensive set of hardware events, e.g. branch mis-predictions and instruction retirements, to recognise standard benchmarking and cryptographic library functions. This includes various signing, verification and hash functions, and ciphers in numerous modes of operation. Three architectures are evaluated using off-the-shelf Intel/X86-64, ARM, and RISC-V CPUs. Next, we show that several known CVE-numbered OpenSSL vulnerabilities can be detected using HPC differences between patched and unpatched library versions. Further, we demonstrate that standardised cryptographic functions within ARM TrustZone TEE applications can be recognised using non-secure world HPC measurements, applying to platforms that insecurely perturb the performance monitoring unit (PMU) during TEE execution. High accuracy was achieved in all cases (86.22–99.83%) depending on the application, architectural, and compilation assumptions. Lastly, we discuss mitigations, outstanding challenges, and directions for future research.

Index Terms—Side-channel analysis, hardware performance counters (HPCs), reverse engineering.

1 Introduction

Modern central processing units (CPUs) support a range of hardware performance counters (HPCs) for monitoring run-time memory accesses, pipeline events (e.g. instruction retired), cache hits, clock cycles, and more. Today's CPUs may support very few HPCs—under 10 on constrained microcontroller units—to over 100 events on Intel and AMD server chips [1], [2]. Originally intended for optimisation and debugging purposes, HPC events have found a myriad of security applications. For example, measurement sources for cache attacks [3]; intrusion detection [4], [5]; malware detection [6], [7], [8]; maintaining control-flow integrity [9]; and reverse engineering proprietary CPU features [10], [11]. While the security implications of high-resolution HPCs have been acknowledged [12], [13], [14], they remain widely available on commercial platforms.

In this paper, we explore a novel application where measurements are analysed *en masse* from multiple counters in order to identify executed program functions. A generic supervised learning workflow is developed, where target functions are classified using events collected before and after their invocation. Analysing exposed functions in this way can help ameliorate time-consuming binary patching and reverse-engineering, e.g. to implement precise code triggers, which is a major challenge in related work [15].

To this end, we present the results of a three-part investigation. Firstly, §3 presents a foundational study on the feasibility of identifying functions from only HPC measurements, using a standard benchmarking suite (MiBench [16]) and four popular cryptography libraries (WolfSSL, Intel's Tinycrypt, Monocypher, and LibTomCrypt). Our approach identifies functions with 48.29%–83.81% accuracy (unprivileged execution) and 86.22%–97.33% (privileged), depending on the target architecture (X86-64/Intel, ARM Cortex-A, and RISC-V). Moreover, we analyse correlations of HPCs and use model inspection techniques to gauge their relative importance. From this, we distill a reduced set of the most effective counters for facilitating generalisation to other platforms that do not support a wide range of HPCs.

After this, §4 explores an offensive use case for detecting patches of known vulnerabilities. This has applications as a reconnaissance method during security evaluations, where we show how several OpenSSL (`libcrypto`) micro-architectural vulnerabilities can be recognised with 0.889–0.998 F_1-score (89.58%–99.83% accuracy). Next, §5 investigates how a malicious spy process may use HPC measurements to recognise cryptographic algorithms executed by a trusted application (TA) within an ARM TrustZone-based trusted execution environment (TEE). Using OP-TEE—an open-source GlobalPlatform-compliant TEE—and a comprehensive set of GlobalPlatform TEE Client API [17] functions, the spy can recognise victim TA functions with 95.50% accuracy. Finally, §6 presents a security analysis, mitigations, and problems for future research.

1.1 Threat Model

We consider an attacker, A, that aims to identify particular algorithms under execution given only high-level function calls and limited knowledge of its implementation. This is often the case when analysing software with no debug symbols, function/variable names, and optimisation and code obfuscation techniques that inhibit readability. An example is the analysis of pre-existing compiled shared libraries on a target system. For instance, Windows dynamic link libraries (DLLs) and Linux shared objects, where the source code is

• *All authors are of Royal Holloway, University of London, Egham, Surrey, United Kingdom. E-mail: carlton@linux.com.*

inaccessible but where a spy application may link to and call functions of interest. A second example is TEE applications, e.g. ARM TrustZone TAs, where only high-level functions calls are exposed to untrusted world software [17]. A's aim is to recognise functions from only CPU HPC measurements taken prior to and following their invocation. This requires kernel-mode code execution to access a full range of HPCs; however, we also explore cases where only user-mode counters are used. A is also assumed to possess oracle access for collecting HPC measurements from idempotent functions, without restrictions on the number of permitted invocations. In our approach, we use a model trained on known HPC-function mappings, i.e. on another system under A's control, which is used for identifying unpatched functions, insecure cryptographic functions, and other tasks on the target.

1.2 Contributions

This paper presents the following contributions:

- The design and evaluation of a generic approach for function recognition using HPC measurements. We develop a test-bed of standard cryptographic and non-cryptographic algorithms taken from widely used cryptographic libraries and MiBench [16] using three off-the-shelf RISC-V, ARM and X86-64 (Intel) platforms. Different privilege levels and compiler optimisations on overall performance are also examined. A feature importance analysis is conducted for determining a strong minimal set of HPCs towards facilitating generalisation.
- Methods and results of two use cases: ① detecting unpatched versions of OpenSSL for several CVE-numbered vulnerabilities; and ② recognising GlobalPlatform cryptographic functions executing in OP-TEE, a GlobalPlatform-compliant ARM TrustZone TEE implementation. This is a *passive* vector that is difficult to mitigate in software against privileged adversaries. For both, experimental results indicate that HPCs can effectively recognise target program functions. Our approach aims to offer an alternative to orthogonal methods for reverse-engineering and vulnerability detection, like those requiring physical access [18], [19] and static analysis and symbolic execution [20], [21].

2 Background

This section discusses related literature and background information on CPU performance events.

2.1 Related Work

HPCs have been used in various security applications, particularly as precise measurement sources for micro-architectural side-channels, like speculative execution and cache timing attacks [3], [22], [23]. A significant body of work has also studied HPCs for malware detection in static and online environments, including rootkit, cryptocurrency miner, and ransomware detection [6], [7], [8]. A general approach instruments target binaries to acquire measurements from the CPU's performance monitoring unit (PMU), which are then used to build custom statistical and machine learning models from known malware/safeware samples.

For intrusion detection, Eunomia [5] traps and analyses sensitive syscalls during suspicious program execution. A trusted monitor analyses the preceding HPC measurements and permits/denies the call to prevent code injection, return-to-libc, and return-oriented programming (ROP) attacks. Xia et al. [9] tackle address control-flow integrity using Intel's Branch Trace Store (BTS), an Intel PMU buffer for storing control transfer events (e.g. jumps, calls, returns, and exceptions). The work builds legal sets of target addresses offline, after which branch traces of suspect applications are compared at run-time. Payer [4] proposed HexPADS, which examines performance events of target processes for detecting Rowhammer, cache attacks, and cross-VM address space layout randomisation (ASLR) breakages.

Copos and Murthy [24] developed a fuzzer that uses HPCs to build valid inputs for closed binaries, where binaries are instrumented for measuring the instruction retirement HPC before executing the program under different inputs. Control flow changes are detected for valid inputs using differences in instruction counters. Spisak [25] developed a kernel-mode rootkit family that uses PMU interrupts to trap system events, such as syscalls. Interestingly, it is shown that TrustZone TAs can perturb the PMU on some consumer devices. Malone et al. [26] investigated static and dynamic software integrity verification using install-time *vs.* run-time HPC measurements. Measurements from six HPCs are presented using four test programs; however, performance results are not given using standard evaluation metrics.

For reverse engineering, Maurice et al. [10] use per-slice PMU access counter measurements to determine the cache slice assigned to a last-level cache (LLC) complex address for enabling cross-core LLC cache attacks. Helm et al. [11] use HPCs for understanding Intel's proprietary DRAM mapping mechanism for translating physical addresses to physical memory channels, banks, and ranks. Similarly, the work uses differences in per-channel PMU transfer counters while accessing different known physical addresses.

Physical attacks have been explored in work with similar applications but under different attack models. Robyns et al. [18] use a convolutional neural network (CNN) to detect eight cryptographic operations using electromagnetic (EM) emissions from a NodeMCU microcontroller, achieving 96% accuracy. Wilt et al. [19] use a similar CNN-based approach for OS and malware detection with ∼84–99% accuracy using EM radiation. Moreover, static analysis and symbolic execution have attracted significant attention for vulnerability detection, achieving 84%–100% accuracy under laboratory conditions, albeit with well-studied problems of model generalisability and state space explosion [20], [21].

2.2 Performance Monitoring Units (PMUs)

Performance counters are available on all major CPU architectures within hardware PMUs, enabling the collection of detailed events with negligible overhead. HPCs are configured and accessed through special-purpose registers that update during execution. While different CPU architectures may count the same types of events, their availability and accessibility can differ significantly. We briefly describe the mechanics of PMUs on X86-64, ARM, and RISC-V.

2.2.1 X86

The Intel PMU, introduced on Pentium CPUs, provides non-configurable registers for tracking fixed events and several programmable registers per logical core [1]. Intel Core CPUs support four general-purpose programmable registers and three fixed-function registers tracking elapsed core cycles, reference cycles, and retired instructions. Programmable registers are configured to simultaneously monitor one of >100 performance events on the Intel Xeon and Core architectures, such as branch mis-predictions and hits at various cache hierarchy levels. PMUs implement configuration and counter registers as model-specific registers (MSRs), accessible using the `RDMSR` and `WRMSR` instructions in ring 0 (kernel mode). The `RDPMC` instruction may also be used for accessing PMU counters at lower privilege levels if the `CR4.PCE` control register bit is set. By default, ring 3 (user mode) access is granted to monotonic time-stamp counters (TSC) in a 64-bit register using the `RDTSC` instruction. In addition to precise event counting, event-based sampling is supported for triggering a performance monitoring interrupt after exceeding threshold value (e.g. after n events). AMD CPUs have minimal functional differences to Intel implementations for counting particular events, but do support more programmable events (six *vs.* four) [1], [2], [4].

2.2.2 ARM

The ARM PMU is a ubiquitous extension for ARM Cortex-A, -M, and -R processors. It supports per-core monitoring of similar but generally fewer high-level events to X86 CPUs. The ARM Cortex-A53, for example, contains a smaller cache hierarchy with a shared L2 cache at the highest level, precluding the ability to report L3 instruction or data cache events [13]. Like X86, fixed-function cycle counters are commonplace, and 2–8 general-purpose counters can be programmed to monitor the events using the `MRS` and `MSR` instructions (AArch64). PMU registers can be configured to be accessed at any privilege mode (exception level) using the performance monitors control register (PMCR). Typically only kernel mode (EL3) processes may access PMU registers by default. ARM PMUs may also assert `nPMUIRQ` interrupt signals, e.g. counter overflows, which can be routed to an interrupt controller for prioritisation and masking.

2.2.3 RISC-V

The RISC-V Privileged [27] and Unprivileged [12] ISA specifications define separate instructions for accessing HPCs in different privilege modes. The Unprivileged ISA specifies 32 64-bit instructions for per-core counters in user- and supervisor-modes (U- and S-mode) and fixed-function counters for cycle count (`RDCYCLE`), real-time clock (`RDTIME`), and instruction retirements (`RDINSTRET`). Control and status register (CSR) space is reserved for 29 vendor-specific 64-bit HPC registers (`HPCCOUNTER3`–`HPCCOUNTER31`). The Privileged ISA specifies analogous CSR registers (`MCYCLE`, `MTIME`, `MINSTRET`) accessible only in machine-mode (M-mode), with 32 64-bit registers (`MHPMCOUNTER3`–`MHPMCOUNTER31`) allocated for vendor-specific HPCs. Note that RISC-V embedded systems are expected to possess only M- or M- and U-modes [27], while workstations and servers are expected to support S-mode and the coming hypervisor (H)-mode extensions [12].

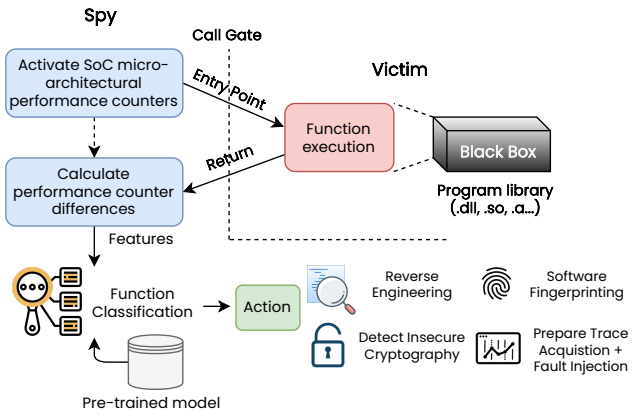


Fig. 1: High-level approach.

3 Function Recognition: A Preliminary Study on X86, ARM, and RISC-V

This section presents a foundational study on recognising a large range of standard functions using HPC measurements. We discuss the methodological approach and evaluated functions, prior to implementation challenges and results.

3.1 Overview

We assume two processes shown in Fig. 1: a *Spy* controlled by the adversary, and a *Victim* that exposes high-level functions. In practical cases, the *Victim* will assume the form of a compiled static or shared library with which the *Spy* is statically or dynamically linked. Our approach then follows two steps: ① HPC measurement (feature) vectors corresponding to each function are assigned labels according to its identifier, which are used for building the model hypothesis. Next, ② uses newly measured values and the model from ① to infer the executed function. We evaluate this using three platform architectures:

- **X86-64**. Dell Latitude 7410 with 8GB RAM and an Intel i5-10310U: 1.70GHz 64-bit quad-core, eight hyper-threads, and 256kB L1, 1MB L2, and 6MB L3 caches. Ubuntu 20.04 LTS was used with Linux kernel v5.12.
- **ARM**. Raspberry Pi 3B+ with 1GB SDRAM and a Broadcom BCM2837 system-on-chip (SoC): 1.4GHz 64-bit quad-core ARM Cortex-A53 CPU, with 32kB L1, 32kB L2, and no L3 cache. Raspbian OS was used, based on Debian 11/Bullseye, with Linux kernel v5.15.
- **RISC-V**. SiFive HiFive Rev. B with a FE310-G002 SoC: 320MHz 32-bit single-core CPU with RV32IMAC ISA support, 6kB L1 instruction cache, and 16kB SRAM. Supports privileged (M-) and unprivileged execution (U-mode). SiFive's Freedom E SDK was used as a hardware abstraction layer for application development.

We used PAPI [28] on our X86-64 and ARM devices, which provides portable HPC measurement acquisition using the Linux *perf* subsystem. CPUs often expose extremely precise access to micro-architectural events, including pipeline- and DRAM controller-specific events, and proprietary features (e.g. Intel TSX), thereby preventing cross-platform compatibility. To overcome this, PAPI implements micro-architectural dependent code and exposes

TABLE 1: HPC events used on a per-device basis.

Method	Description	Device X86-64	Device ARM	Device RISC-V
RDTSCP	Cycle count since a reset.	✓	✗	✗
L1_DCM	L1 data cache misses.	✓	✓	✗
L1_ICM	L1 instruction cache misses.	✓	✓	✓
L1_TCM	L1 total cache misses.	✓	✗	✗
L1_LDM	L1 load misses.	✓	✗	✗
L1_DCA	L1 data cache accesses.	✗	✓	✗
L1_STM	L1 store misses.	✓	✗	✗
L2_DCM	L2 data cache misses.	✓	✓	✗
L2_ICM	L2 instruction cache misses.	✓	✗	✗
L2_TCM	L2 total cache misses.	✓	✗	✗
L2_LDM	L2 load misses.	✓	✗	✗
L2_DCR	L2 data cache reads.	✓	✗	✗
L2_STM	L2 store misses.	✓	✗	✗
L2_DCA	L2 data cache accesses.	✓	✓	✗
L2_ICR	L2 instruction cache reads.	✓	✗	✗
L2_ICH	L2 instruction cache hits.	✓	✗	✗
L2_ICA	L2 instruction cache accesses.	✓	✗	✗
L2_TCA	L2 total cache accesses.	✓	✗	✗
L2_TCR	L2 total cache reads.	✓	✗	✗
L2_TCW	L2 total cache writes.	✓	✗	✗
L3_TCM	L3 total cache misses.	✓	✗	✗
L3_LDM	L3 load misses.	✓	✗	✗
L3_DCA	L3 data cache accesses.	✓	✗	✗
L3_DCR	L3 data cache reads.	✓	✗	✗
L3_DCW	L3 data cache writes.	✓	✗	✗
L3_ICA	L3 instruction cache accesses.	✓	✗	✗
L3_ICR	L3 instruction cache reads.	✓	✗	✗
L3_TCA	L3 total cache accesses.	✓	✗	✗
L3_TCR	L3 total cache reads.	✓	✗	✗
L3_TCW	L3 total cache writes.	✓	✗	✗
CA_SNP	Requests for a cache snoop.	✓	✗	✗
CA_SHR	Requests for exclusive access to a shared cache line.	✓	✗	✗
CA_CLN	Requests for exclusive access to a clean cache line.	✓	✗	✗
CA_ITV	Requests for cache line intervention.	✓	✗	✗
TLB_DM	Data TLB misses.	✓	✓	✗
TLB_IM	Instruction TLB misses.	✓	✓	✗
PRF_DM	Data pre-fetch cache misses.	✓	✗	✗
MEM_WCY	Cycles stalled for memory writes.	✓	✗	✗
STL_ICY	Cycles with no instruction issue.	✓	✗	✗
FUL_ICY	Cycles with maximum instruction issue.	✓	✗	✗
BR_UCN	Unconditional branch instructions.	✓	✗	✗
BR_CN	Conditional branch instructions.	✓	✗	✗
BR_TKN	Conditional branches taken.	✓	✗	✗
BR_NTK	Conditional branch instructions not taken.	✓	✗	✗
BR_MSP	Conditional branch mispredictions.	✓	✓	✗
BR_PRC	Conditional branches correctly predicted.	✓	✗	✗
TOT_INS	Instructions completed.	✓	✓	✗
LD_INS	Load instructions.	✓	✓	✗
SR_INS	Store instructions.	✓	✓	✗
BR_INS	Branch instructions.	✓	✓	✗
RES_STL	Cycles stalled on any resource.	✓	✗	✗
TOT_CYC	Total cycles executed.	✓	✓	✗
LST_INS	Load and store instructions executed	✓	✗	✗
SP_OPS	Single-precision floating point operations.	✓	✗	✗
DP_OPS	Double-precision floating point operations.	✓	✗	✗
VEC_SP	Single precision SIMD instructions.	✓	✗	✗
VEC_DP	Double precision SIMD instructions.	✓	✗	✗
RDCYCLE	U-mode reference cycle count.	✗	✗	✓
RDINSTRET	U-mode instructions retired.	✗	✗	✓
RDTIME	U-mode real-time counter	✗	✗	✓
MCYCLE	M-mode cycle counter	✗	✗	✓
MINSTRET	M-mode instructions retired.	✗	✗	✓
MTIME	M-mode real-time counter.	✗	✗	✓

X86-64: Intel i5-10310U; ARM: ARM Cortex-A53; RISC-V: SiFive FE310.
Blue cells indicate user-mode counters; yellow denote privileged counters.

common high-level HPC events, which we used for portability. Table 1 shows the availability of these events on Intel, ARM and RISC-V. PAPI was compiled and dynamically linked with our benchmarking application that implemented the functions under test (§3.2) on X86-64 and ARM. For collecting unprivileged events, our application executed in user-mode only. To access privileged HPCs, we set `kernel.perf_event_paranoid=-1` prior to execution to allow the reporting of kernel-mode counters in user space from Linux *perf*. For RISC-V, the aforementioned assembly instructions were directly used for accessing HPC registers (§2.2.3) in M- (privileged) and U-modes (unprivileged).

3.2 Target Procedures

We developed a test-bed comprising 64 functions listed in Table 2, incorporating the MiBench benchmarking suite with common embedded systems procedures [16], alongside modern cryptographic algorithms. For the latter, we used an extensive range of common functions available through

TABLE 2: Test-bed target functions.

Non-cryptographic Functions		
1. Solve Cubics	8. Bsearch	15. Euler
2. Integer Sqrt	9. CRC32	16. Simpson
3. Angle Convert	10. BWT	17. Root Finding
4. Qsort	11. Matrix Mul.	18. XOR-Shift
5. Dijkstra	12. Bit Ops.	19. Base64 Encode
6. PBM Search	13. Gaussian Elim.	20. Base64 Decode
7. FFT	14. Fibonacci	21. Entropy
Cryptographic Functions		
22-23. AES-ECB (E+D)	40-41. 3DES (E+D)	56. GMAC
24-25. AES-CBC (E+D)	42-43. GOST (E+D)	57. Poly1305
26-27. AES-CTR (E+D)	44-45. Ed25519 (S+V)	58. MD2
28-29. AES-GCM (E+D)	46-47. ECDSA (S+V)	59. MD4
30-31. ChaCha20 (E+D)	48. X25519	60. MD5
32-33. ChaCha20+Poly1305 (E+D)	49. ECDH	61. SHA-1
34-35. Speck (E+D)	50-53. RSA (E+D, S+V)	62. SHA-256
36-37. PRINCE (E+D)	54. DH	63. SHA-3
38-39. DES (E+D)	55. HMAC	64. BLAKE2

E+D: Encrypt and decrypt. S+V: Sign and verify.

WolfSSL in addition to reference implementations of GOST, Speck, and PRINCE. Our benchmarking tool used random input buffers in the range [8B, 32B, 256B, 512b, 1024kB, 2048kB, 4096kB] for encryption, signing, MAC, and hashing algorithms. This was used to avoid potential HPC measurement biases if only a fixed-size input was used for functions that accepted an arbitrary byte buffer and its length. Random keys were generated for all symmetric algorithms and keyed MACs (e.g. AES in all modes, ChaCha20, Prince, DES and HMAC). Similarly, random public-private key pairs and secret and public values were generated for asymmetric and key exchange algorithms respectively (e.g. ECDSA, X25519, ECDH, and RSA). HPC measurements were not inclusive of this process. Fresh initialisation vectors (IVs), counter values and nonces were also generated where applicable (e.g. ChaCha20 and AES-CTR). RSA operations were split approximately equally using a random padding scheme and key length. RSAES-OAEP or RSAES-PKCS#1v1.5 were used for encryption/decryption, and RSASSA-PSS or RSASSA-PKCS#1v1.5 for signing/verification under 1024-, 2048-, 3072-, and 4096-bit key lengths. Similarly, ECDSA and ECDH used a random curve from P-256, P-384, and P-521, while AES used 128-, 196, and 256-bit key lengths. For ease of implementation, corresponding inverse operations (e.g. verification for signing) were called with the same parameters immediately following the original operation.

3.3 Methodology

For each available HPC event in Table 1, we collected measurements from 10,000 invocations for each of the 64 test-bed functions. For instance, Qsort on RISC-V was invoked 70,000 times in total to collect the necessary measurements (10,000 invocations $\times$ 7 available HPCs). These N-dimensional feature vectors ($N = 49$, X86-64; $N = 13$, ARM; $N = 7$, RISC-V) were mapped to the class label of the procedure ($[0, 64) \in \mathbb{N}$), i.e. $\sim$31.3M invocations in total (X86-64), 8.3M (ARM), and 4.4M (RISC-V). We then formed two data sets: **A** and **B**, representing HPCs available in unprivileged and privileged modes respectively.

Each data set was split into training and test sets using a 80:20 ratio, before applying Z-score normalisation to produce features with zero mean and unit variance.

Given the relatively large and balanced many-class classification problem, conventional classification accuracy was used as the evaluation metric. A preliminary experiment initially explored a basic template approach using four similarity metrics—Euclidean distance, Minkowski distance, cross-correlation, and covariance—to classify test vectors against the training set. However, the approach produced unpromising results: <40% accuracy in the best cases using branch predictions and total instructions. We thus resorted to training nine supervised classifiers: naïve Bayes (NB), logistic regression (LR), linear discriminant analysis (LDA), decision tree (DT), gradient boosting machines (GBM), random forest (RF), support vector machine (SVM), and a multi-layer perceptron (MLP). Hyper-parameter optimisation was conducted using an exhaustive grid search and k-fold ($k = 10$) cross-validation (CV), with the best performing CV classifier scoring the test set.

3.4 Implementation Challenges

Using HPCs requires careful thought to avoid measurement bias and other pitfalls. Some phenomena have been investigated in related literature [29], [30], while others have attracted little attention, like cache warming and compiler optimisations. We discuss these challenges forthwith.

3.4.1 HPC Accuracy

Run-to-run variations in HPC measurements is a challenge on commercial CPUs for program analysis [29], [30]. Counter measurements are not perfectly replicated between sequential program executions, with a 0.5%–2% deviation observed in counter values using standard benchmarks [30]. This arises principally from:

- *Event non-determinism*. External events, particularly hardware interrupts and page faults, can cause small deviations in vulnerable HPC events, e.g. load and store counters, which are unpredictable and difficult to reproduce. Another source of variation is the pipeline effects arising, for instance, from out-of-order execution. This can also perturb absolute counter values, although the effects can be negligible if the monitoring of very short instruction sequences is avoided [13].
- *Overcount*. CPU implementation differences and errata can overcount events. For example, instructions retired events can be overcounted by exceptions and pseudo-instructions where micro-coded instructions differ to the instructions that are actually executed. Specifically, X87/SSE exceptions, OS lazy floating point handling, and `fldcw`, `fldenv`, `frstor`, `maskmovq`, `emms`, `cvtpd2pi`, `cvttpd2pi`, `sfence`, `mfence` instructions are known overcount sources [30].

The effects of non-determinism cannot be entirely eliminated. Rather, we model interactions from many individual measurements (millions of features) as a mitigation against single, run-to-run variations. This assumes the ability to measure large numbers of target functions idempotently, i.e. their behaviour changes insignificantly between invocations, and without restricting the number of permitted calls (see §1.1). Importantly, we do not rely upon exact measurements, but only that the deviations between different function executions confer enough discriminitive power for classification. Moreover, we use measurements from *multiple* counters to maximise the discriminative benefits of disparate event types, which also minimises potential issues (e.g. overcounting bias) of using any single counter. X86 CPUs can also measure the number of hardware interrupts—a known source of non-determinism—which was used as a feature (Table 1).

3.4.2 Function Implementation Variations

It is important to discern between multiple implementations of the *same* algorithm. Statistical models may fail to generalise when presented with HPC events of alternative libraries that, while similar algorithmically, contain code differences that can affect HPC measurements (e.g. API calling conventions). Considering this, we ported additional implementations from Monocypher (for ChaCha20, Poly1305, Ed25519, X25519, BLAKE2), Intel's Tinycrypt (AES, ECDSA, ECDH, HMAC, SHA-256), and LibTomCrypt (AES, MD2, MD4, MD5, SHA-1, SHA-256, 3/DES, RSA, DH, GMAC, HMAC) on our target platforms.[1] For these functions, measurements were taken by cycling through each implementation for each invocation, and mapping them to the same label. For instance, MD5 measurements from LibTomCrypt *and* WolfSSL implementations were assigned the same label.

3.4.3 Cache Warming

Preliminary experiments showed that cache-based events, such as L1 and L2 misses, and their correlated values—discussed further in §3.6.1—were markedly higher during initial function executions. This subsided after several executions per function in order to converge to stability (<3% requiring an average of 9.7 executions, X86-64; 7.0, ARM; 7.3, RISC-V). We hypothesise that cache warming was partially responsible, which has not been in related literatur on HPC-based security applications. Within the context of this work, the reliance on large numbers of measurements renders this effect negligible (<0.5%). However, the reader is warned of potential bias in constrained environments where only a limited number of measurements can be collected.

3.4.4 Measuring Multiple Performance Events

Today's PMUs support a small number of programmable counter registers, typically 4–8 depending on the architecture [1], [2]. Consequently, measuring several events requires: ① Using a single counter register per execution; ② Batching HPC measurements, with the batch size equalling the number of supported counter registers; or ③ Using software multiplexing provided by some tools, e.g. PAPI, where counters are time-shared over many performance events. We used ① as a conservative method at the cost of collection time. With ②, collection time could be reduced by a constant factor (of the max. supported counters), while ③ minimises collection time at the cost of accuracy [28].

3.4.5 Compiler Optimisations

Modern compilers improve performance and code size at the expense of compilation time and debuggability; for

1. The choice of these libraries was to construct a common test-bed; very few cryptographic libraries (e.g. OpenSSL and GnuTLS) currently offer 32-bit RISC-V MCU support.

TABLE 3: Classification accuracy (Intel i5-10310U; in %; data set A = unprivileged HPCs, B = privileged).

		Classifier								
	Data set	NB	LR	kNN	DT	RF	GBM	LDA	SVM	MLP
A	O0	49.30	50.24	60.24	59.20	**66.31**	65.79	61.56	54.44	51.83
	Os	49.22	50.63	58.30	64.46	66.07	**66.56**	61.87	62.43	54.62
	O3	48.48	49.30	59.08	**60.27**	59.81	49.71	35.57	45.46	47.28
	Mixed	37.21	32.80	**48.29**	48.03	47.67	40.52	26.30	45.39	37.02
B	O0	98.95	96.47	89.25	99.53	**99.70**	99.02	94.31	99.04	86.55
	Os	93.03	89.27	87.71	95.89	95.99	**97.49**	88.29	92.35	90.06
	O3	87.55	86.08	82.75	92.01	**98.51**	83.78	74.44	80.53	84.60
	Mixed	84.78	73.95	85.14	95.40	**97.33**	91.01	80.86	83.78	85.95

TABLE 4: Classification accuracy (ARM Cortex-A53; privileged counters).

	Classifier								
Data set	NB	LR	kNN	DT	RF	GBM	LDA	SVM	MLP
O0	85.65	87.27	82.34	96.87	96.74	**96.81**	80.42	83.65	84.01
Os	77.91	77.45	82.70	90.31	88.59	**91.32**	86.28	84.13	84.30
O3	79.21	82.99	82.85	88.90	**91.40**	89.43	74.37	87.62	74.82
Mixed	73.97	70.08	80.35	88.96	88.99	**90.68**	74.00	80.37	69.09

TABLE 5: Classification accuracy (SiFive E31 SoC; data set A = unprivileged HPCs, B = privileged).

		Classifier								
	Data set	NB	LR	kNN	DT	RF	GBM	LDA	SVM	MLP
A	O0	76.09	74.59	78.30	85.55	**89.04**	89.00	87.10	81.67	80.28
	Os	72.84	73.02	79.61	86.47	83.40	**86.91**	80.35	77.53	77.54
	O3	72.88	74.94	83.99	85.10	**87.42**	86.80	76.75	74.32	75.02
	Mixed	66.87	66.23	83.60	82.16	**83.81**	75.68	70.07	68.15	67.22
B	O0	82.83	83.46	82.02	90.87	**93.34**	91.99	86.57	81.05	83.30
	Os	80.19	78.67	79.44	87.23	**90.04**	90.03	75.11	72.84	75.56
	O3	81.01	76.48	**88.29**	86.32	86.37	87.41	79.85	74.32	80.20
	Mixed	76.74	74.02	69.81	84.59	**86.22**	86.21	70.03	71.40	70.90

example, using loop unrolling, if-conversions, tail-call optimisation, inline expansion, and register allocation. This is an important consideration given the difficulty of knowing *a priori* the exact compilation parameters of target software, e.g. a dynamic library. Some compiler optimisations can have a material effect on HPC measurements for the same program: Choi et al. [31], for instance, showed that if-conversions can reduce branch mis-predictions by 29% on Intel CPUs, which would reflect in branch-related HPCs.

To address this, we compiled the test-bed and collected measurements from test devices under different GCC optimisation options: disabling optimisation (O0, the default setting); minimising code size (Os); and maximum optimisation (O3) that sacrifices compile time and memory usage for performance. The reader is referred to the GCC documentation for a comprehensive breakdown of the optimisations utilised with each flag [32]. To emulate environments where compiler optimisations are unknown, a mixed data set was formed by concatenating and randomly shuffling measurement vectors collected under each flag.

3.5 Results

The data collection and training processes were orchestrated by a Python script using Scikit-Learn [33] on a workstation with an Intel i7-6700k CPU (quad-core at 4.0GHz) and 16GB RM, taking approximately 14 hours (data collection) and 30 hours (training). Classification results for each platform and GCC compilation setting are presented in Tables 3 (X86-64), 4 (ARM), and 5 (RISC-V). In general, functions could be classified from HPC values with considerable effectiveness, albeit with notable differences with respect to the architecture and privilege mode. Worst-case performances occurred where only unprivileged counters were used with the mixed GCC data sets, i.e. 48.29% (X86-64; one HPC) and 83.81% (RISC-V; three HPCs). Mixed compilation parameters generally correlated with a significant accuracy degradation of 2–16% depending on the architecture and privilege mode. In contrast, the best cases corresponded to scenarios where all HPCs were used for specific compilation settings: 97.33%–99.70% (X86-64), 90.68%–96.81% (ARM), and 86.22–93.34% (RISC-V). We can tentatively conclude that using more HPCs confers greater discriminitive power during classification, the availability of which are maximised during privileged execution. Further, tree-based models and ensembles tended to perform best out of all evaluated classifiers (18/20) with RF classifiers the best-performing (11/20).

Confusion matrices were produced for examining class-level performance weaknesses using the mixed GCC data sets and the best-performing classifiers (see Fig. 2 for X86-64; Appendix A for ARM and RISC-V). Confusion is observed in functions with structural similarities when fewer counters are used (Fig. 2b). For instance, the encryption and decryption functions for ChaCha20-Poly1305, DES, 3DES, and AES in certain modes of operation (e.g. CTR and CBC), and RSA encryption, decryption, signature and verification. These errors resolved when more HPCs were available, where greatly reduced classification accuracy was observed.

3.6 Feature Importance

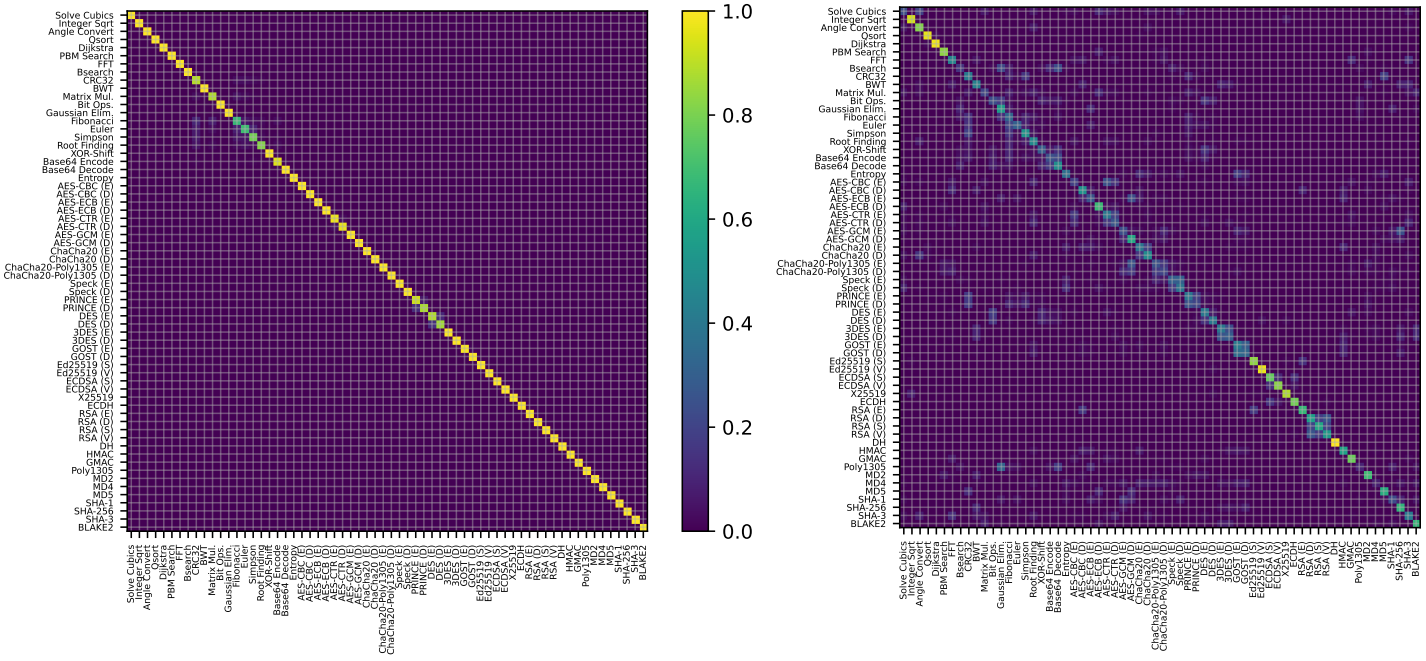
An important consideration is the contribution of each HPC feature to the classification process. If only a small set of HPCs is needed to train high accuracy models, then certain implementation difficulties can be avoided. Minimising the number of hardware counters is desirable for two reasons:

1) *Reproducibility*: Using all of the counters on a given architecture risks depending on redundant HPCs that are unavailable on other architectures, particularly older and constrained platforms. ARM and RISC-V microcontroller and IoT SoCs, for instance, contain fewer measurable events relative to workstation- and server-grade X86-64 CPUs [1], [13], [27].
2) *Performance*: Removing redundant features, or those with low predictive power, can offer training and classification performance benefits due to the curse of dimensionality. (Dimensionality reduction methods have been applied in HPC malware classification literature, e.g. principal component analysis [6], [7], but this does not directly reduce the number of HPCs used at source).

We note that HPC-based research tends to rely on complex, non-linear models, such as random forests and gradient boosting machines, to model high-dimensional data [6] (also observed in §3.5). Unfortunately, these models have decision processes that are inherently difficult to interpret for determining the most effective features, prompting the development of model explanation methods [34], [35].

3.6.1 HPC Correlation Analysis

As a first step, we examined the correlations between HPC measurements on each platform. It is intuitive that certain



(a) Privileged HPCs.

(b) Unprivileged HPCs.

Fig. 2: Normalised confusion matrices for X86-64 (X-axis: Predicted value, Y-axis: True value).

features may exhibit significant collinearities (a historically successful method of identifying redundant features [36]). For example, the total number of load/store instructions (`LST_INS`) is directly related to the number of load instructions (`LD_INS`). Likewise, processes that frequently access temporally and spatially dislocated data will cause cache misses and, thus, more cache data writes. Fig. 4 presents pairwise correlation matrices using the correlation coefficients of HPC tuples.

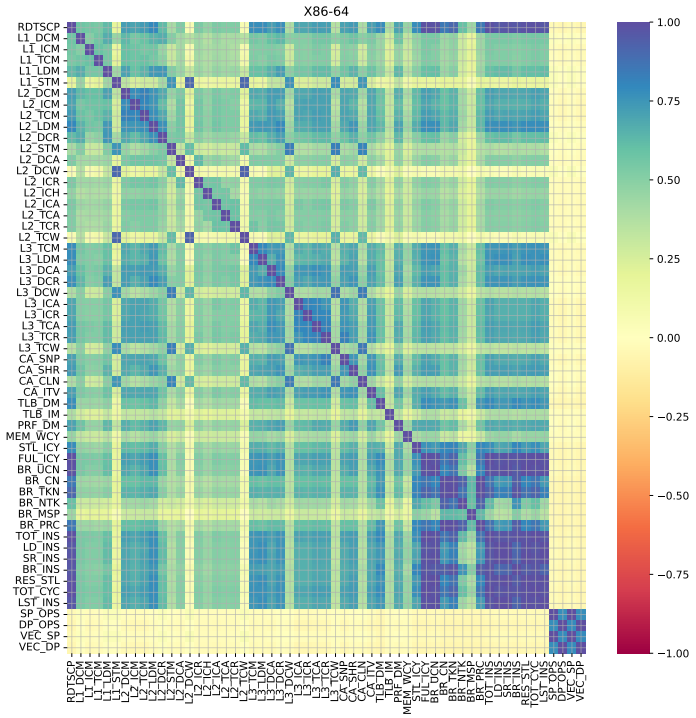
The results show strong correlations between many HPC pairs. On X86-64, the cycle and time-stamp counters (`TOT_CYC` and `RDTSCP`) are >0.95 correlated with the total instruction (`TOT_`, `LD_`, `SR_`, `BR_` and `LST_INS`) and branch counters (`BR_TKN`, `_NTK`, and `_PRC`). A similar pattern was found for ARM. This is intuitively unsurprising considering that larger, longer-running functions will likely contain more run-time branches and memory accesses. Strong correlations are also seen between cache misses in lower cache hierarchy elements and accesses in higher ones; for example, L2 data accesses and L1 data misses (`L2_DCA` and `L1_DCM`, 0.96; X86-64). On ARM, L1 *instruction* cache misses are strongly correlated (0.98) with L2 *data* accesses (`L1_ICM` vs. `L2_DCA`). In the absence of a dedicated L2 instruction cache [13], this indicates that the L2 data cache is used as a *de facto* instruction cache, echoing existing work on using HPCs to uncover latent SoC properties [10], [11].

It is also seen that cache misses in last-level caches (LLC)—L3 for X86-64 and L2 for ARM (`L3_TCM` and `L2_DCM` respectively)—have no correlations with other HPCs. PAPI monitors *CPU-level* events, and the LLC represents the final unit before external memories are accessed. Similar patterns exist for TLB data misses (`TLB_DM`) on ARM and instruction misses (`TLB_IM`) on X86-64 and ARM; and the use of vector and floating point operations on X86-64 (`SP_OPS`, `DP_OPS`, `VEC_SP`, `VEC_DP`), which only have correlations with each other. On RISC-V, strong correlations (>0.97) were observed between M-mode counters of their U-mode counterparts, e.g. `RDINSTRET` vs. `MINSTRET`. This is interesting from a security perspective: unprivileged processes can use HPCs with potentially the same power as privileged processes. It also provides insight into why classification results for privileged HPCs were only marginally different (∼4%) from unprivileged HPCs in §3.5.

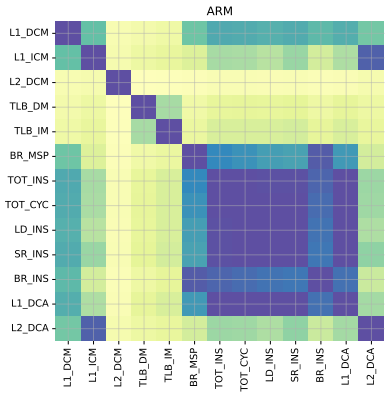
3.6.2 *Shapley Additive Explanations*

While useful for understanding multicollinearity, correlations do not show the extent to which particular HPCs contribute to classification decisions, thus requiring model inspection methods to understand their relative importance. To this end, we employed SHAP (SHapley Additive exPlanations), a unified framework for interpreting complex model predictions, which has been applied to Android malware classification [37] and intrusion detection [38]. SHAP uses a co-operative game-theoretic approach for assigning feature importances of a machine learning model prediction function, f, using Shapley values [34]. SHAP explains f's decisions as the sum of results, $\phi_i \in \mathbb{R}$, of feature subsets being included in a conditional expectation, $E[f(x)|x_S]$ (S being a subset of model features and x the feature vector of the instance to be explained). SHAP scores combine conditional expectations with the Shapley value of a feature value, ϕ_i—corresponding to its contribution to the payout (prediction)—which are calculated using Eq. 1.

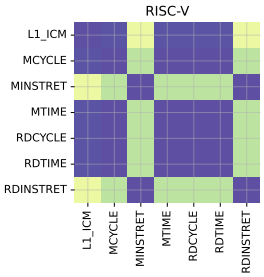
$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(M - |S| - 1)!}{M!} [f_x(S \cup \{i\}) - f_x(S)] \quad (1)$$



(a) X86-64 HPCs.

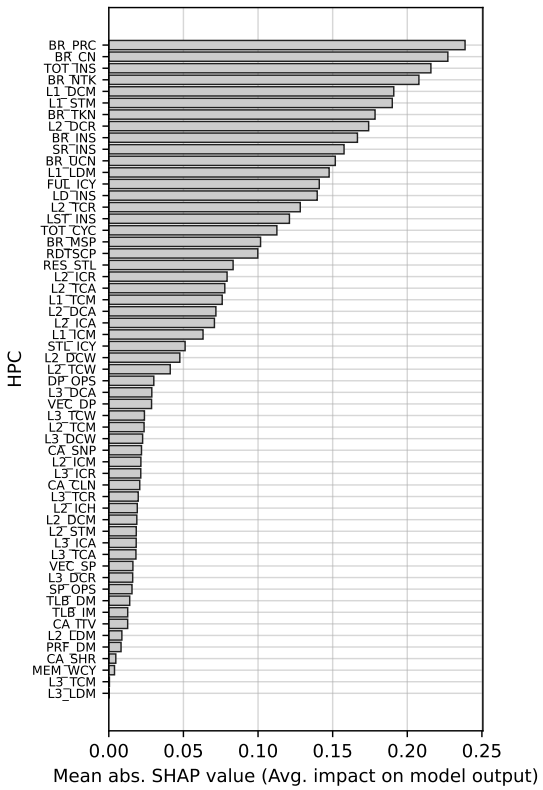


(b) ARM HPCs.

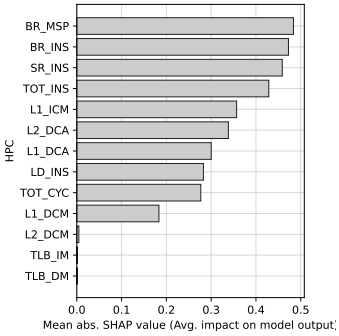


(c) RISC-V HPCs.

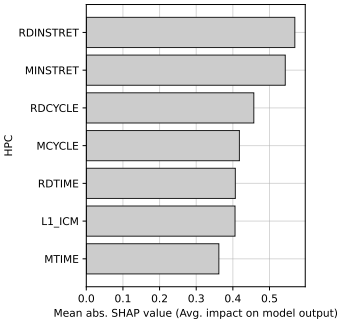
Fig. 3: Correlation matrices for each test-bed device.



(a) X86-64 HPCs.



(b) ARM HPCs.



(c) RISC-V HPCs.

Fig. 4: Rank-ordered mean absolute SHAP values.

Where M is the number of features and N is the set of all input features. The average absolute Shapley values per feature are computed across the entire data set and rank-ordered to find the global feature importances. SHAP values are more consistent with human intuition than alternative approaches, like LIME [39] and DeepLIFT [35]; are model-agnostic; and are not liable to the properties of HPC measurements. (Other techniques, e.g. mean decrease in Gini impurity for tree models, can be misleading with high cardinality or continuous features [40]—an inherent property of HPCs).

We computed and rank-ordered the SHAP values for each platform HPC using the best-performing classifiers from §3.5 under the mixed GCC data sets (Fig. 4). Certain HPCs had a significant impact on classification decisions across all platforms. Branch counters formed the top two ARM HPCs (`BR_MSP`, `BR_INS`) and six of the top 10 X86-64 counters (`BR_PRC`, `BR_CN`, `BR_NTK`, `BR_TKN`, `BR_INS`, `BR_UCN`). The RISC-V platform could not measure branch events. Instruction counters also ranked highly, representing the top two RISC-V HPCs (`RDINSTRET`, `MINSTRET`), three of the top five ARM HPCs (`BR_INS`, `SR_INS`, `TOT_INS`), and three of the top 10 X86-64 HPCs (`TOT_INS`, `BR_INS`, `SR_INS`). We observe that cache events ranked relatively poorly, particularly TLB data and instruction misses (X86-64 and ARM), and LLC events (ARM L2 and X86-64 L3).

3.6.3 *Feature Elimination*

SHAP values gauge the relative contribution of features, but do not directly determine an effective minimal set of HPCs required for strong model performance. Therefore, we investigated how model accuracy fluctuated by systematically including particular hardware counters. Here, the best performing models from §3.5 were retrained using the top N features from the SHAP analysis under the same classifier

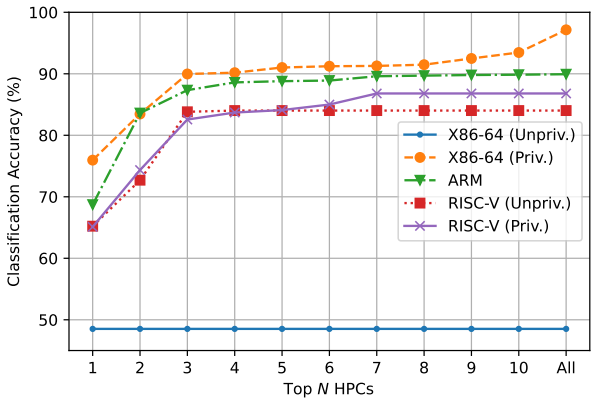


Fig. 5: Accuracy using the top N SHAP-valued HPCs.

hyper-parameters. Values of N were evaluated in the range 1–10 and compared with using all available HPCs.

The results are shown in Fig. 5 for each platform using HPCs accessible in unprivileged and privileged mode. Evidently, using all HPCs bestows *some* accuracy benefits, but only a subset were necessary for achieving close results. Using all 49 X86-64 counters, for example, delivered a ∼5% improvement over using the top 10 from Fig. 4a. The top three HPCs achieved <9% of the final accuracy on X86-64 (`BR_PRC`, `BR_CN`, `TOT_INS`) while only two ARM counters were needed to achieve this (`BR_MSP`, `BR_INS`). On RISC-V, the accuracy benefits decreased to ∼5% using three counters (`RDINSTRET`, `MISNTRET`, `RDCYCLE`) versus all seven.

Our results generally indicate that a large range of black-box functions can be classified effectively based on their HPC values—up to 97.33%—across three major CPU platforms. This varies significantly according to privilege mode and counter availability on our chosen platforms. In the coming sections, we explore applications of this generic approach in the domains of vulnerability detection within OpenSSL and function recognition within TEE applications.

4 Identifying Known Security Vulnerabilities in OpenSSL

Cryptographic libraries are often deployed as static or shared libraries in end-user applications. OpenSSL, for instance, is deployed as `libssl` and `libcrypto`, with the latter implementing dependent cryptographic algorithms which may be used in isolation. By default, both libraries are compiled and named using major and minor version numbers, e.g. 1.0, 1.1, and 3.0. Incremental sub-versions are also released after remedying known security vulnerabilities, known as *lettered* releases.

In this use case, we examine the extent to which vulnerable `libcrypto` lettered versions can be detected using HPC measurements from invoking affected cryptographic functions. Specifically, we examine the extent to which OpenSSL vulnerabilities can be identified using HPC differences in calls to patched and unpatched `libcrypto` functions. This can be used as an exploratory technique for isolating useful attack vectors in the dependencies of application binaries. Broadly, the attacker is assumed to possess the ability to: ① instrument the target binary to measure function calls to the dependent library; or ② compile and link his/her own measurement harness to invoke functions in the dependency.

4.1 Methodology

Our developed approach follows three phases:

1) *Preliminary vulnerability identification.* We comprehensively examined the OpenSSL vulnerability disclosure announcements[2] to identify vulnerabilities that cause internal micro-architectural state changes, but whose effects are not immediately observable (e.g. does not induce a crash, infinite loop, or returns certain function values). Vulnerable/patched functions were located using NIST's National Vulnerability Database (NVD), which aggregates technical write-ups, third-party advisories, and, importantly, commit-level patch details for CVEs. In total, we successfully identified six vulnerabilities, which may be used for DSA, RSA, and ECDSA private key recovery. The descriptions, CVE numbers, and commit IDs are given in Appendix B.
2) *Collecting labelled data samples*. For each vulnerability, we collected HPC measurements of 10,000 executions using all available counters (privileged and unprivileged). We measured offending `libcrypto` functions of the same major and minor version, but from different *lettered* versions before and after the patch was implemented. For example, for a function patched in `v1.1.0f`, measurements were taken from the preceding `v1.1.0a-e` (unpatched) and successive versions `v1.1.0f-n` (patched). This required compiling and linking our PAPI measurement harness against multiple individual `libcrypto` lettered versions. In contrast to §3, which considered functon recognition as a multi-class problem, vulnerability detection is treat as a *binary* problem, where measurement vectors were assigned labels in the range $[0, 1] \in \mathbb{N}$ (0 = patched, 1 = unpatched).
3) *Model selection and evaluation*. Following the same method as §3.3, multiple models were trained using an 80:20 training-test set ratio, 10-fold cross-validation, and exhaustive grid search for hyper-parameter optimisation. In addition to classification accuracy, precision, recall, and F1-score metrics were employed for evaluating binary classification performance. This is important when considering the unbalanced nature of vulnerability detection in this context. Measurement data sets of vulnerabilities remedied in early OpenSSL lettered versions, e.g. `1.1.0b`, will contain far fewer 'unpatched' labels than those in later versions, thus necessitating evaluation metrics that consider relevance.

4.2 Results and Analysis

We applied the methodology to known vulnerabilities within `libcrypto` using the same target devices from §3.[3] The classification results are presented in Tables 6 and 7 for X86-64 and ARM respectively for each vulnerability. Our approach identified OpenSSL vulnerabilities with high accuracy: 89.5% in all cases and 93%+ accuracy in all but one

2. https://www.openssl.org/news/vulnerabilities.html
3. OpenSSL does not support RISC-V at the time of publication.

TABLE 6: OpenSSL CVE identification results (X86-64).

CVE ID	Operation	Pr.	Re.	F1	Acc.
CVE-2018-5407	ECC scalar mul.	0.915	0.977	0.945	94.67
CVE-2018-0734	DSA sign	0.997	0.998	0.998	99.83
CVE-2018-0735	ECDSA sign	0.974	0.950	0.962	97.50
CVE-2018-0737	RSA key gen.	0.892	0.900	0.896	90.25
CVE-2016-2178	DSA sign	0.985	0.978	0.981	98.75
CVE-2016-0702	RSA decryption	0.940	0.938	0.939	95.92

Pr.: Precision, Re.: Recall, F1: F1-score, Acc.: Accuracy (in %).

TABLE 7: OpenSSL CVE identification results (ARM).

CVE ID	Operation	Pr.	Re.	F1	Acc.
CVE-2018-5407	ECC scalar mul.	0.891	0.975	0.931	93.25
CVE-2018-0734	DSA sign	0.976	0.995	0.985	99.00
CVE-2018-0735	ECDSA sign	0.978	0.910	0.943	96.33
CVE-2018-0737	RSA key gen.	0.882	0.896	0.889	89.58
CVE-2016-2178	DSA sign	0.954	0.938	0.946	96.42
CVE-2016-0702	RSA decryption	0.939	0.873	0.904	93.83

case across both architectures. Likewise, precision (0.892–0.997, X86-64; 0.882–0.978, ARM), recall (0.900–0.998, X86-64; 0.873–0.995, ARM) and F1 scores (0.896–0.998, X86-64; 0.889–0.985, ARM) were consistently high, demonstrating effectiveness when identifying feature vectors corresponding to unpatched instances.

It is noteworthy that some differences exist between each identified CVE, particularly CVE-2018-0737, which has significantly lower accuracy (4–5%) than the next worst performing CVE. On closer inspection, we noticed a broad correlation between classification performance and relative patch complexity. This is not surprising: security-related patches with significant code additions and/or deletions will cause deterministic effects in HPC measurements. As an elementary example, additional conditional statements will correlate with increased values of branch-related, cycle, and instruction counters. The CVE-2018-0737 patch—the worst performing vulnerability—changed only two lines of code (LOC) between lettered OpenSSL versions for setting constant-time operation flags for RSA primes (commit 6939eab03a6e23d2bd2c3f5e34fe1d48e542e787[4]).[5] Compare this to CVE-2018-0734—detected with the highest accuracy—which made 19 LOC changes, including the declaration of new variables, conditional statements, internal function calls, and more (commit 8abfe72e8c1de1b95f50aa0d9134803b4d00070f[6]).

This raises a research question about the *granularity* with which counters can reliably detect arbitrary code changes. The known effects of non-determinism—see §3.4 and Das et al. [29]—and pipeline execution on measurement noise [13] provide an undetermined lower bound. Yet, this has not been answered in related literature, which we pose as an obvious gap for future research. Notwithstanding, our approach shows that HPC measurements of function calls can detect known vulnerabilities using off-line analysis.

4. https://github.com/openssl/openssl/commit/6939eab03a6e23d2bd2c3f5e34fe1d48e542e787

5. LOC is an illustrative proxy of program complexity; a single line may induce complex control flows, with significant effects on HPCs.

6. https://github.com/openssl/openssl/commit/8abfe72e8c1de1b95f50aa0d9134803b4d00070f

5 Recognising Cryptographic Algorithms in a Trusted Execution Environment (TEE)

The previous section showed how some security vulnerabilities can be detected within compiled OpenSSL (libcrypto) libraries. The general approach is extendable to recognising functions executing within trusted execution environment (TEE) applications. TEE applications typically expose high-level APIs for enabling untrusted applications to interact with secure world services [13], [14], [17]. In this section, we examine an application of our approach to recognise standardised cryptographic functions within ARM TrustZone TEE applications from a malicious, non-secure world process.

5.1 ARM TrustZone and the ARM PMU

ARM TrustZone partitions platform execution into 'secure' (SW) and 'non-secure' (NS) worlds, with the aim of protecting SW service services from kernel-level, non-secure world software attacks. SW execution is isolated by setting the NS-bit by the secure monitor at the highest ARM exception (privilege) level (EL3), which is added to cache tags and propagated through system-on-chip bus transactions, e.g. for accessing sensitive peripheral controllers. Interactions between the normal and secure worlds are conducting using ARM secure monitor calls (SMC) for entering secure monitor mode. NS world applications invoke TA functions using a pre-defined interfaces specified by the TA developer, as standardised by the GlobalPlatform Client API [17]. Notably, TEE TAs are usually provisioned in encrypted form, which are subsequently loaded from flash memory during the device's secure boot sequence using a firmware-bound key. This occurs *before* loading any untrusted world binaries, rendering direct inspection of TA binaries tremendously difficult, even from privileged NS world execution [14], [15].

Recall from §2.2.2 that the ARM PMU manages performance events on ARM Cortex-A platforms. Ideally, PMU interrupt events should be suppressed during secure world execution to prevent sensitive micro-architectural state changes being measurable from malicious non-secure world processes. However, enabling SW PMU events is commonly used in pre-release testing environments for TEE debugging and optimisation (a non-invasive method under the ARM debugging architecture [41]). Whether or not SW PMU events are enabled can be determined by querying the non-invasive and secure non-invasive flags (NIDEN and SPNIDEN) of the ARM DBGAUTHSTATUS debug register. If NIDEN or SPNIDEN are set, then PMU events are counted in the non-secure and secure worlds respectively.

Importantly, Spisak [25] showed that SPNIDEN was still enabled in some consumer devices following release, including the Amazon Fire HD 7″ tablet and Huawei Ascend P7. This was followed by Ning and Zhang [41] who examined 11 mobile, IoT and ARM server platforms, showing that only three devices had correctly unset SPNIDEN prior to consumer release. Vulnerable devices included the Huawei Mate 7, Raspberry Pi 3B+, Xiaomi Redmi 6, and the Scaleway ARM C1 Server. The use of insecurely configured ARM PMU events during secure world execution has been exploited on consumer devices for building rootkits using PMU interrupts [25], and cross-world covert channels on an

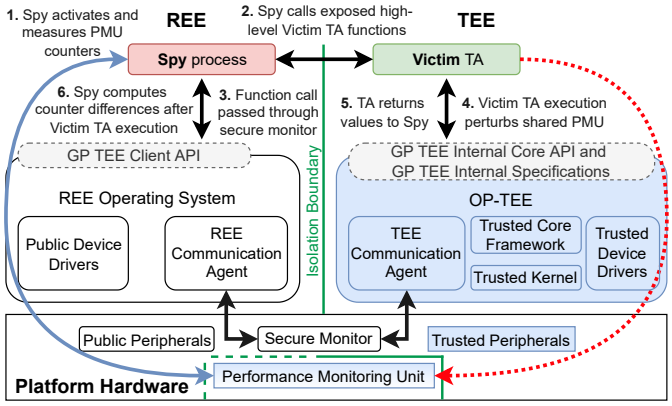


Fig. 6: TEE side-channel set-up using a shared PMU.

undisclosed Samsung Tizen TV (ARM Cortex-A17, ARMv7) and HiKey board (ARM Cortex-A53, ARMv8) [42].

5.2 Methodology

Using the knowledge that consumer devices may fail to suppress SW PMU interrupts, we developed a test-bed for investigating the extent to which cryptographic algorithms within secure world TAs can be identified from non-secure world processes. OP-TEE[7] was leveraged to this end—an open-source, GlobalPlatform-compliant TEE reference implementation—on our Raspberry Pi 3B+. Two applications were developed, which are illustrated in Fig. 6:

- *Spy* (non-secure world). A privileged application that uses the GlobalPlatform Client API to invoke TEE functions in the *Victim*. The same PAPI test harness used in §3 and §4 measured HPC values immediately before and after TA command was invoked, i.e. `TEEC_InvokeCommand()` in GlobalPlatform Client API nomenclature. The *Spy* may be unprivileged if the PMU is configured to enable measurements from EL0.
- *Victim* (secure world TA). An emulation of a TEE-based key management application that exposes four high-level functions for ① signing (`TA_SIGN`), ② verifying (`TA_VERIFY`), ③ encrypting (`TA_ENCRYPT`), and ④ decrypting (`TA_DECRYPT`) inputs provided by the *Spy* on demand. These utilised fresh TEE-bound keys which were generated at random on a per-session basis.[8]

The aim, similar to §3, is to identify the precise algorithm used by the *Victim* from PMU measurements before and after its invocation by the *Spy*. The *Spy* is given only the aforementioned high-level functions for signing, verification, encryption, and decryption. Secure world cryptographic functions are implemented in OP-TEE using LibTomCrypt whose functions are wrapped by the GlobalPlatform Internal Core API [17]. The *Victim* was developed to call an extensive range of GlobalPlatform Internal Core API functions implemented by the OP-TEE OS core. For calculating performance results, the *Victim* TA also accepted a given Internal Core API algorithm ID, which was used for labelling HPC vectors. The full list of analysed cryptographic algorithms is given in Table 8, covering all available modes of operation and padding schemes where applicable.

7. https://op-tee.org

8. Under the GlobalPlatform TEE architecture, a non-secure client application invokes one or more *commands*, e.g. signing or decryption, of a target TA within a single *session*.

TABLE 8: *Victim* TA algorithms and key sizes (bits).

Method	GlobalPlatform Internal Core API ID	Key Sizes
TA_SIGN and TA_VERIFY		
DSA	TEE_ALG_DSA_SHA1	512, 1024
	TEE_ALG_DSA_SHA224	2048
	TEE_ALG_DSA_SHA256	2048, 3072
ECDSA	TEE_ALG_ECDSA_P192	192
	TEE_ALG_ECDSA_P224	224
	TEE_ALG_ECDSA_P256	256
	TEE_ALG_ECDSA_P384	384
	TEE_ALG_ECDSA_P521	512
RSA	TEE_ALG_RSASSA_PKCS1_V1_5_MD5	
	TEE_ALG_RSASSA_PKCS1_V1_5_SHA1	
	TEE_ALG_RSASSA_PKCS1_V1_5_SHA224	
	TEE_ALG_RSASSA_PKCS1_V1_5_SHA256	
	TEE_ALG_RSASSA_PKCS1_V1_5_SHA384	
	TEE_ALG_RSASSA_PKCS1_V1_5_SHA512	1024, 2048
	TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA1	3072, 4096
	TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA224	
	TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA256	
	TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA384	
	TEE_ALG_RSASSA_PKCS1_PSS_MGF1_SHA512	
TA_ENCRYPT and TA_DECRYPT		
AES	TEE_ALG_AES_CBC_NOPAD	
	TEE_ALG_AES_CCM	
	TEE_ALG_AES_CTR	128, 196
	TEE_ALG_AES_ECB_NOPAD	256
	TEE_ALG_AES_GCM	
DES	TEE_ALG_DES_ECB_NOPAD	64
	TEE_ALG_DES_CBC_NOPAD	
3DES	TEE_ALG_DES3_ECB_NOPAD	128, 192
	TEE_ALG_DES3_CBC_NOPAD	
RSA	TEE_ALG_RSAES_PKCS1_V1_5	
	TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA1	
	TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA224	1024, 2048
	TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA256	3072, 4096
	TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA384	
	TEE_ALG_RSAES_PKCS1_OAEP_MGF1_SHA512	

Using our previously developed PAPI test harness, HPC measurements of 1,000 invocations of each algorithm were collected from the non-secure world using all available HPCs on our platform. This was repeated for 100 sessions, with the data collected afterwards for off-line analysis (100,000 vectors per algorithm; 3.4M in total). The measurement vectors were labelled with respect to each GlobalPlatform Internal Core algorithm identifier, i.e. with the label $[0, 34) \in \mathbb{N}$. Like §3, key sizes were not fixed; a random key size was set during the algorithm's run-time allocation prior to its execution. It is important to note that OP-TEE TAs are cross-compiled using a GCC-based toolchain, rendering them vulnerable to the compilation biases (§3.4.5). To address this, we evaluated the GCC optimisation flags from §3 in order to assess the effects of different optimisation levels on classification performance.

5.3 Results

After retrieving the data files from the test platform, the same procedure was followed as in the previous sections. The data file was divided into training and test sets using an 80:20 ratio before applying exhaustive grid search with 10-fold cross-validation to select the best performing classifier.

TABLE 9: Classification accuracy using *Spy* (non-secure world) HPC feature vectors from *Victim* (TEE) execution.

	Classifier								
Dataset	**NB**	**LR**	**kNN**	**DT**	**RF**	**GBM**	**LDA**	**SVM**	**MLP**
O0	83.99	90.12	88.07	94.48	**96.13**	90.76	92.02	88.54	87.90
Os	84.61	87.92	84.68	90.72	94.07	**96.02**	95.51	85.81	86.75
O3	83.18	92.49	93.19	89.58	93.33	**97.45**	79.66	94.35	87.71
Mixed	82.89	85.27	92.83	**95.50**	94.17	86.93	82.55	86.99	90.82

The final accuracy was calculated using the performance of the best performing cross-validation classifier on the aforementioned test set. Results of this analysis are given in Table 9. The results generally reflect those in §3 and §4: algorithm recognition can be achieved with high accuracy using PMU values perturbed by a secure world TA that is measured by a non-secure world application. In the best case, 95.50% classification accuracy (DT) was achieved for the mixed GCC data-set, increasing slightly to best cases of 96.13% (RF), 96.02% (GBM), and 97.45% (GBM) for the O0, Os, and O3 data sets respectively.

6 Evaluation

This section analyses mitigations, limitations, and challenges of the work presented in this paper.

6.1 Analysis and Mitigations

Exploiting PMUs as a side-channel medium has been acknowledged by CPU architecture designers and specification bodies. ARM concede that counters are a potential side channel for leaking confidential information, and issue secure development guidance for preventing TEEs from perturbing HPC measurements during TA execution [13], [14]. Likewise, the RISC-V Unprivileged specification states that *"Some execution environments might prohibit access to counters to impede timing side-channel attacks"* [12] (Chapter 10, p. 59). Despite this, PMUs are still widely accessible with privileged access and are still perturbed by TEE TAs on some consumer devices [25], [41], [42].

If attackers are assumed to possess only user-mode execution, then certain system-level countermeasures can be deployed. On X86-64, the `CR4.TSD` and `CR4.PCE` control registers can be set and unset to prevent the reading of time-stamp (`RDTSC`) and programmable PMU counter values (`RDPMC`) respectively. On Linux devices, the `kernel.perf_event_paranoid` flag can be set to a non-zero value to prevent user-space processes from accessing PMU values through the *perf* subsystem. Alternatively, as a common but blunt countermeasure, *perf* can be disabled at installation time for providing access to high-resolution CPU events. It is worth stating that performance counters have many legitimate uses, including application benchmarking and debugging, which would be prevented by this mitigation.

Side-channel resistance has been studied extensively in the context of avoiding time- and data-dependent code. Cache-based timing attacks, in particular, have prompted the development of resistant cryptographic algorithms (e.g. bit-sliced AES implementations). In recent years, attacks that leverage side-effects of transient execution have compounded these issues [22], [23]. From the results in this work, we emphasise that focussing on time-, cache-based, or branch prediction-based side-channel mitigations are insufficient for protecting against the presented methods. Ideally, library and TEE TA developers must consider a larger range of measurable performance events, such as instruction retirements, TLB hits/misses, load/store counts, pre-fetch misses, and memory writes. While certain *individual* events have been exploited and mitigated, *multiple* events can be leveraged to bypass countermeasures against any particular micro-architectural side-channel approach.

For software-based mitigations, inspiration can be taken from Li and Gaudiot [43] who posed the challenge of HPC-based speculative execution attack detection in the presence of 'evasive' adversaries. State-of-the-art accuracy of Spectre detection models declined by 30%–40% after introducing instructions that mimicked benign programs. Thus, one countermeasure is to introduce micro-architectural obfuscation by randomly and significantly perturbing the PMU during the execution of sensitive functions. The injection of well-crafted injection was also suggested by Liu et al. [44] for countering ARM cache-based side-channels and by Carrelli et al. [45] against certain timing attacks. However, we such stress that noise must apply to *many* HPC events, not just cache- or timing-based counters. How this can be feasibly and effectively achieved is posed as an open challenge.

Attention is also drawn to countermeasures when deploying Intel Software Guard eXtensions (SGX) enclaves. SGX applications can be built using the 'anti side-channel interference' (ASCI) feature, which suppresses interrupts to Intel PMUs upon entry to production enclaves [1], thus preventing their use as a side-channel for inspecting enclave contents. Unless developers explicitly and negligently opt-out of ASCI, then production enclaves are strengthened significantly to the attacks described in this paper. Likewise, we reiterate best-practice guidelines to device manufacturers to prevent PMU events being raised during secure world execution by securely configuring the PMU control and debug registers upon TEE entry.

6.2 Challenges, Limitations, and Practicability

Using HPCs to classify program functions faces some interesting challenges that were considered outside the scope of this work. Firstly, we investigated programs with few levels of intermediate abstraction: using C programs on a bare-metal microcontroller (RISC-V) or with a single host OS (Debian-based Linux; ARM and X86-64). Instrumenting and measuring programs using HPCs faces difficulties in virtualised environments sharing a single set of CPU counters; for instance, within virtual machines (VMs) and containers with OS-level virtualisation (e.g. Docker). Programs written in interpreted languages and those with just-in-time (JIT) compilation, e.g. Java, also pose known challenges for side-channel analysis [15]. Further research is required to correctly account for the additional noise from multiple users, processors, garbage collectors, etc. on a single PMU.

Secondly, a significant number of possible implementations variations may be encountered in reality on an arbitrary platform. The variations were extensive but not exhaustive; for example, the use of hardware implementations; different RNG sources; expanded cryptographic libraries, e.g. Crypto++ and Bouncy Castle; and alternative

TEE implementations were not evaluated. The scope of this work was not to provide an comprehensive analysis of these possibilities. Rather, we aimed to provide the first investigation into using HPCs for function recognition, while providing a detailed understanding of HPC correlations, relative classification contributions, and the applications to vulnerability detection and TA analysis. We also wish to briefly note that the recognition of *arbitrary* functions cannot be practically achieved considering that it may not terminate, reflecting the halting problem. For constrained cases, like recognising encryption functions from a set of potential library candidates, the data acquisition and classification stages remained feasible in nature: $<$3 minutes per feature vector in the worst cases of unoptimised RSA encryptions and DH key exchanges on our RISC-V microcontroller.

Thirdly, in a practical scenario, the generic approaches in §3 and §5 would require classifiers to be trained and transferred to a device under test. TrustZone software binaries, for instance, are typically encrypted and authenticated during secure boot sequences on consumer devices. Moreover, TrustZone OSs are notoriously closed-source and closed-access, preventing users from installing custom TAs after deployment for acquiring ground-truth labels. A future research direction is to explore the efficacy of *transfer learning* by training a model initially in an accessible white-box environment where samples can be reliably labelled before transposing it to black-box cases. Although GlobalPlatform TEE APIs *specify* a standard set of algorithms, one challenge with this approach is that their *implementations* may differ during the transfer between white-box and black-box, OEM implementations. While we evidenced that different library implementations may be classified under a single label in §3, this remains untested in a transfer learning environment.

Lastly, we stress that our approach was evaluated on regular programs. Obfuscated binaries, particularly those that maliciously perturb HPCs to thwart PMU-based methods, can exhibit more complex behaviours that may pose generalisation challenges, which we defer to future research.

7 Conclusion

This paper developed, implemented and evaluated a generic approach to black-box program function recognition. Extending work from related HPC research, we explored a side-channel approach in which micro-architectural events are analysed in bulk using a generic machine learning workflow. We then presented a three-part evaluation of this approach: ① A preliminary study that examined classifying functions from widely used benchmarking suites and cryptographic libraries; ② detecting several known, CVE-numbered vulnerabilities within OpenSSL; and ③ cryptographic function recognition within ARM TrustZone. The approaches achieved 86.22%–99.83% accuracy depending on the target architecture and application. We showed how functions are recognisable in a relatively large, multi-class problem space. Furthermore, we demonstrated how OpenSSL lettered versions containing security vulnerabilities can be identified with high accuracy (0.889–0.998 F_1-score; 89.58%–99.83% accuracy). We then presented results from a further use case for recognising functions in a reference open-source ARM TrustZone TEE implementation with high accuracy (95.50%–97.45%) using a comprehensive range of GlobalPlatform TEE API functions. The results our work are broadly commensurate with alternative approaches, albeit with different adversarial models, including physical EM analysis for detecting cryptographic operations, malware detection, and OS detection (84–99%) and static analysis and symbolic execution (84%–100%). Rather than supplanting these methods, we hope that our approach offers new thinking for software analysis using HPCs.

We posit that focussing on well-known side-channel vectors—cache accesses, timing differences, and branch predictions—-are insufficient for engineering implementations which are resistant to the methods presented in this paper. Focus must directed to a wider range of micro-architectural events simultaneously—TLB misses, instruction retirements, clock cycles, pre-fetch events, and others—rather than prescribed events popularised in related work. We also re-emphasise best-practice guidelines for configuring PMUs to avoid exposing TEE side-channels to untrusted applications. Further work is needed to leverage more sophisticated attack scenarios, e.g. interpreted languages and transferring models *between* consumer devices, but we present evidence that HPC-based function recognition is effective on today's major CPU architectures.

Acknowledgments

This work received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 883156 (EXFILES).

References

[1] Intel, Inc., "Intel 64 and IA-32 architectures software developer's manual combined volumes 3A, 3B, 3C, and 3D: System programming guide," 2022.

[2] AMD, Inc., "AMD64 architecture programmer's manual volume 2: System programming," 2021.

[3] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2008, pp. 59–67.

[4] M. Payer, "HexPADS: a platform to detect 'stealth' attacks," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2016, pp. 138–154.

[5] L. Yuan, W. Xing, H. Chen, and B. Zang, "Security breaches as PMU deviation: detecting and identifying security attacks using performance counters," in *2nd Asia-Pacific Workshop on Systems*, 2011, pp. 1–5.

[6] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *ACM Asia Conference on Computer and Communications Security*, 2018, pp. 457–468.

[7] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *ACM Asia Conference on Computer and Communications Security*, 2017, pp. 483–493.

[8] M. Alam, S. Bhattacharya, S. Dutta, S. Sinha, D. Mukhopadhyay, and A. Chattopadhyay, "RATAFIA: ransomware analysis using time and frequency informed autoencoders," in *IEEE Int'l Symposium on Hardware Oriented Security and Trust*. IEEE, 2019.

[9] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *IEEE Int'l Conference on Dependable Systems and Networks*. IEEE, 2012.

[10] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Int'l Symposium on Recent Advances in Intrusion Detection*. Springer, 2015.

[11] C. Helm, S. Akiyama, and K. Taura, "Reliable reverse engineering of Intel DRAM addressing using performance counters," in *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2020.

[12] RISC-V Foundation, "The RISC-V instruction set manual volume I: Unprivileged ISA," 2019.

[13] ARM, "ARM Cortex-A53 MPCore processor: Technical reference manual," 2014.

[14] ——, "Secure development guidelines," 2021, https://trustedfirmware-a.readthedocs.io/en/latest/process/security-hardening.html.

[15] C. Shepherd, K. Markantonakis, N. van Heijningen, D. Aboulkassimi, C. Gaine, T. Heckmann, and D. Naccache, "Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis," *Computers & Security*, vol. 111, Dec 2021.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *4th Annual IEEE International Workshop on Workload Characterization*. IEEE, 2001, pp. 3–14.

[17] GlobalPlatform, "TEE System Architecture (v1.3)," 2022.

[18] P. Robyns, M. Di Martino, D. Giese, W. Lamotte, P. Quax, and G. Noubir, "Practical operation extraction from electromagnetic leakage for side-channel analysis and reverse engineering," in *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2020, pp. 161–172.

[19] M. L. Wilt, M. M. Baker, and S. J. Papadakis, "Toward an RF side-channel reverse engineering tool," in *IEEE Physical Assurance and Inspection of Electronics*. IEEE, 2020, pp. 1–7.

[20] B. Zhang, C. Feng, B. Wu, and C. Tang, "Detecting integer overflow in Windows binary executables based on symbolic execution," in *17th IEEE/ACIS Int'l Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 2016.

[21] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, 2021.

[22] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 564, 2017.

[23] C. Li and J.-L. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters," in *30th Int'l Symposium on Computer Architecture and High Performance Computing*. IEEE, 2018.

[24] B. Copos and P. Murthy, "Inputfinder: Reverse engineering closed binaries using hardware performance counters," in *5th Program Protection and Reverse Engineering Workshop*, 2015.

[25] M. Spisak, "Hardware-assisted rootkits: Abusing performance counters on the ARM and x86 architectures," in *10th USENIX Workshop on Offensive Technologies*, 2016.

[26] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs?" in *6th ACM Workshop on Scalable Trusted Computing*. ACM, 2011.

[27] RISC-V Foundation, "The RISC-V instruction set manual volume II: Privileged ISA," 2021.

[28] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing*. Springer, 2010.

[29] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 20–38.

[30] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *IEEE Int'l Symposium on Performance Analysis of Systems and Software*. IEEE, 2013.

[31] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai, "The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor," in *34th ACM/IEEE Int'l Symposium on Microarchitecture*, 2001.

[32] Free Software Foundation, Inc., "Optimize options (using the GNU Compiler Collection)," 2022, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[34] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, ser. NeurIPS, 2017, pp. 4765–4774.

[35] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *International conference on machine learning*. PMLR, 2017, pp. 3145–3153.

[36] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, The University of Waikato, 1999.

[37] M. Fan, W. Wei, X. Xie, Y. Liu, X. Guan, and T. Liu, "Can we trust your explanations? Sanity checks for interpreters in Android malware analysis," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 838–853, 2020.

[38] M. Wang, K. Zheng, Y. Yang, and X. Wang, "An explainable machine learning framework for intrusion detection systems," *IEEE Access*, vol. 8, pp. 73 127–73 141, 2020.

[39] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you?: Explaining the predictions of any classifier," in *22nd ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*.

[40] C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn, "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC Bioinformatics*, vol. 8, no. 1, pp. 1–21, 2007.

[41] Z. Ning and F. Zhang, "Understanding the security of ARM debugging features," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 602–619.

[42] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn, "Prime+Count: Novel cross-world covert channels on ARM TrustZone," in *34th Annual Computer Security Applications Conference*, 2018, pp. 441–452.

[43] C. Li and J.-L. Gaudiot, "Challenges in detecting an 'evasive spectre'," *IEEE Computer Architecture Letters*, 2020.

[44] N. Liu, W. Zang, S. Chen, M. Yu, and R. Sandhu, "Adaptive noise injection against side-channel attacks on ARM platform," *EAI Endorsed Transactions on Security and Safety*, vol. 6, no. 19, 2019.

[45] A. Carelli, A. Vallero, and S. Di Carlo, "Performance monitor counters: interplay between safety and security in complex cyber-physical systems," *IEEE Transactions on Device and Materials Reliability*, 2019.

Carlton Shepherd received his Ph.D. in Information Security from the Information Security Group at Royal Holloway, University of London, U.K., and his B.S. in Computer Science from Newcastle University, U.K. He is currently a Research Fellow at the Information Security Group at Royal Holloway, University of London, where his research interests centre around the security of trusted execution environments (TEEs), security-enhanced CPU designs, and embedded systems security.

Benjamin Semal received his M.Eng. in Electrical Engineering from Ecole Polytechnique Universitaire of Montpellier, France and M.S. in robotics from Cranfield University, U.K. He then worked as a hardware security analyst at UL Transaction Security. He later joined Royal Holloway, University of London to pursue a Ph.D. in Information Security. His research focusses on side-channel attacks for information leakage in multi-tenant environments. Benjamin now works as a security engineer at SERMA Security & Safety evaluating point-of-sale devices and cryptographic modules.

Konstantinos Markantonakis received his B.S. in Computer Science from Lancaster University, U.K.; and his M.S. and Ph.D. in Information Security, and M.B.A. in International Management from Royal Holloway, University of London, London, U.K. He is currently the Director of the Smart Card and IoT Security Centre. He has co-authored over 190 papers in international conferences and journals. His research interests include smart card security, trusted execution environments, and the Internet of Things.

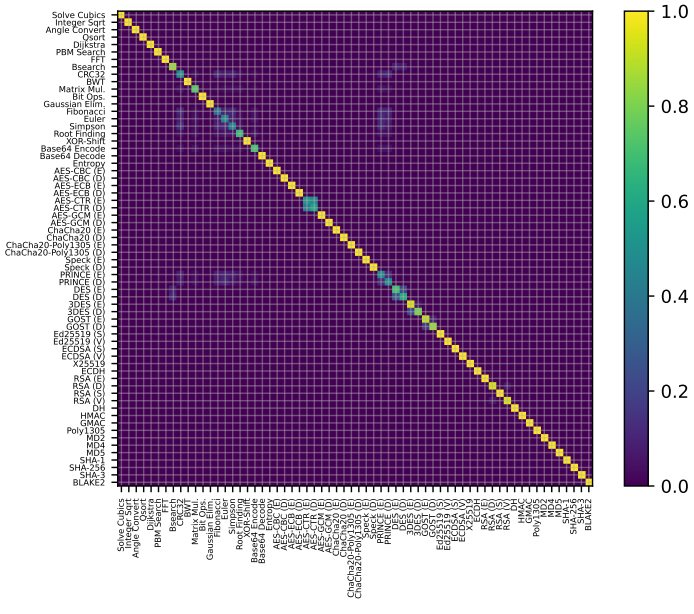
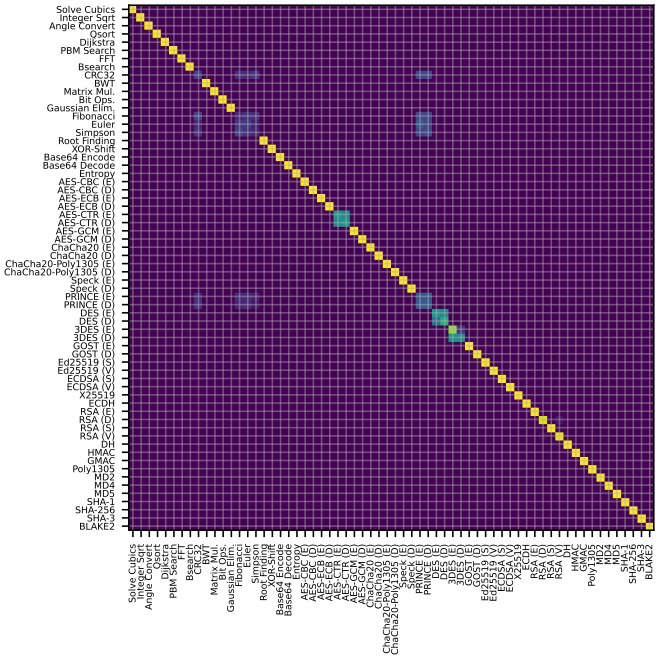
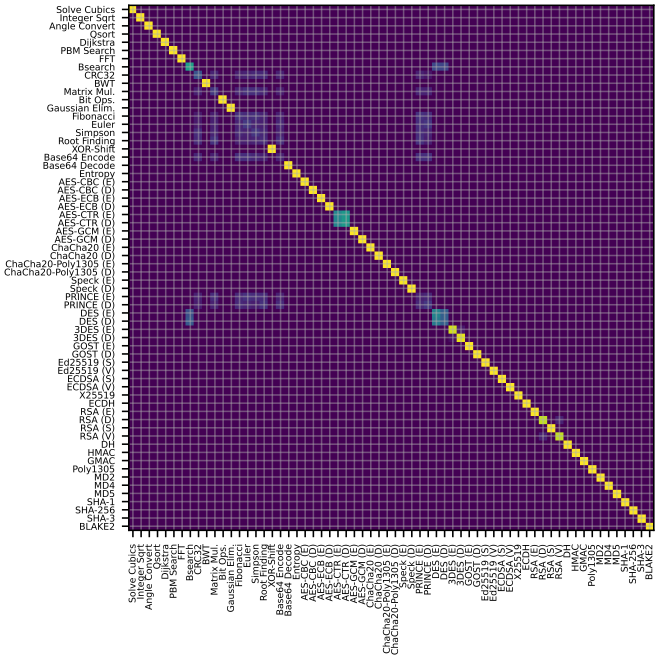


Fig. 7: Normalised ARM confusion matrix.



(a) RISC-V (Privileged).



(b) RISC-V (Unprivileged).

Fig. 8: Normalised RISC-V confusion matrices.

Appendix A
ARM and RISC-V Confusion Matrices

Confusion matrices relevant to §3.5 for ARM and RISC-V function recognition are given in Figs. 7 and 8 respectively.

Appendix B
OpenSSL CVE Descriptions

This appendix provides an overview of the six OpenSSL CVEs explored in §4 as follows:

(V1): `CVE-2018-5407`: *"ECC scalar multiplication, used in e.g. ECDSA and ECDH, has been shown to be vulnerable to a microarchitecture timing side channel attack. An attacker with sufficient access to mount local timing attacks during ECDSA signature generation could recover the private key."* Fixed in OpenSSL v1.1.0i.

(V2): `CVE-2018-0734`: *"DSA signature algorithm has been shown to be vulnerable to a timing side channel attack. An attacker could use variations in the signing algorithm to recover the private key."*. Fixed in v1.1.1a.

(V3): `CVE-2018-0735`: *"The OpenSSL ECDSA signature algorithm has been shown to be vulnerable to a timing side channel attack. An attacker could use variations in the signing algorithm to recover the private key."* Fixed in v1.1.0j.

(V4): `CVE-2018-0737`: *"RSA key generation algorithm has been shown to be vulnerable to a cache timing side channel attack. An attacker with sufficient access to mount cache timing attacks during the RSA key generation process could recover the private key."* Fixed in v1.1.0i.

(V5): `CVE-2016-2178`: *"Operations in the DSA signing algorithm should run in constant time in order to avoid side channel attacks. A flaw in the OpenSSL DSA implementation means that a non-constant time codepath is followed for certain operations. This has been demonstrated through a cache-timing attack to be sufficient for an attacker to recover the private DSA key."* Fixed in v1.0.2i.

(V6): `CVE-2016-0702`: *"A side-channel attack was found which makes use of cache-bank conflicts on the Intel Sandy-Bridge microarchitecture which could lead to the recovery of RSA keys. The ability to exploit this issue is limited as it relies on an attacker who has control of code in a thread running on the same hyper-threaded core as the victim thread which is performing decryptions."* Fixed in v1.0.2g.

APPENDIX C
MODEL HYPER-PARAMETER DETAILS

The model hyper-parameters used in the results in Tables 3–7 and 9 were derived from the ranges presented in Table 10.

TABLE 10: Evaluated model hyper-parameters (Scikit-Learn nomenclature [33]; default values used otherwise).

Method	Hyper-parmeter ranges
NB	Variance smoothing: [1e-1, 1e-3, 1e-5, 1e-7, 1e-9, 1e-11, 1e-13]
LR	C (regularisation): [1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4] Optimisation solver: ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
kNN	N neighbours: [1, 3, 5, 10, 25, 50, 75, 100]
DT	Split criterion: ['gini', 'log_loss', 'entropy'] Max. tree depth: [2, 5, 7, 10, 25, 50, 75, 100, None]
RF	Split criterion: ['gini', 'log_loss', 'entropy'] Max. tree depth: [2, 5, 7, 10, 25, 50, 75, 100, None] N estimators: [5, 10, 25, 50, 100, 250, 375, 500]
GBM	Learning rate: [0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1.0] N estimators: [5, 10, 25, 50, 100, 250, 375, 500] Max. tree depth: [1, 3, 5, 7, 10, 25, 50, 75, 100]
LDA	Optimisation solver: ['svd', 'lsqr', 'eigen']
SVM	Kernel: ['linear', 'poly', 'rbf', 'sigmoid'] C (regularisation): [1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4] γ (kernel coefficient): ['auto', 'scale']
MLP	Hidden layer dimensions: [(50,), (100,), (250,), (50,50), (100,100), (250,250), (50,50,50), (100,100,100), (250,250,250), (100,250,100) (50,100,250,250), (50,50,50,50), (100,100,100,100),(250,250,250,250)] Activation functions: ['identity', 'logistic', 'tanh', 'relu']