IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 769–794. DOI:10.46586/tches.v2024.i2.769-794

High-Performance Design Patterns and File Formats for Side-Channel Analysis

Jonah Bosland[1], Stefan Ene[1], Peter Baumgartner[2] and Vincent Immler[1]

[1] Oregon State University, Corvallis, USA, {boslandj,enes,immlerv}@oregonstate.edu
[2] Independent Security Researcher, Munich, Germany, researchsec@eml.cc

Abstract. Data and instruction dependent power consumption can reveal cryptographic secrets by means of Side-Channel Analysis (SCA). Consequently, manufacturers and evaluation labs perform thorough testing of cryptographic implementations to confirm their security. Unfortunately, the computation and storage needs for the resulting measurement data can be substantial and at times, limit the scope of their analyses. Therefore, it is surprising that only few publications study the efficient computation and storage of side-channel analysis related data.
To address this gap, we discuss high-performance design patterns and how they align with characteristics of different file formats. More specifically, we perform an in-depth analysis of common side-channel analysis algorithms and how they can be implemented for maximum performance. At the same time, we focus on storage requirements and how to reduce them, by applying compression and chunking.
In addition, we investigate and benchmark popular SCA frameworks. Moreover, we propose SCARR, a proof of concept SCA framework based on the file format Zarr, that outperforms all considered frameworks in several common algorithms (SNR, TVLA, CPA, MIA) by a factor of about two compared to the thus far fastest framework for a given profile. Most notably, in *all* tested scenarios, we are faster even *with* file compression, than other frameworks *without* compression. We are convinced that the presented design patterns and comparative study will benefit the greater side-channel community, help practitioners to improve their own frameworks, and reduce data storage requirements, associated costs, and lower computation/energy demands of SCA, as required to perform more testing at scale.

Keywords: Side-Channel Analysis · High-Performance Computing · file format · HDF5 · Zarr · SCARED · LASCAR · ChipWhisperer

1 Introduction

Secure communication and data storage for a wide range of purposes relies on electronic systems that guarantee integrity and confidentiality of their data. This is governed by security standards such as Common Criteria (CC) [The06], FIPS [Nat02], or PSA Certified Security Assessment [PSA22]. Compliant devices are required to implement mechanisms [ABB+20] to prevent extraction of cryptographic secrets. This includes countermeasures to thwart Differential Power Analysis (DPA) [KJJ99] and Electro-Magnetic Analysis (EMA) [QS01; AAR+03; PSQ07; HMH+12; HMH+13]. There are many examples where insufficient protection against these techniques enabled real-world attacks, e.g., [RLM+21; SW12; OP11; ORP13; PEK+09; MS16]. While it should not be used exclusively, Measurements-to-Disclosure (MTD) and the corresponding data complexity is a popular metric to assess the practical difficulty of these attacks [MOP07; PGA+23].

Testing plays an important role during the development and subsequent security evaluation of countermeasures. Typically, a wide range of measurements must be performed

Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

under varying physical conditions such as type of measurement, Region of Interest (ROI) on a chip, equipment used, and different logical inputs (data acquisition patterns) as stimuli. This can result in vast amounts of data in the lower terabyte range with substantial differences in the assessed security levels of different data sets, despite observing the same implementation [MM12; MM13; UHD+17; ISU18; SIU+18; BUS22]. Because of this, some industry practitioners have previously rejected the practicality of some SCA-related developments in academia. These concerns were presented at COSADE 2013 in the talk "*Academic vs. industrial perspective on SCA*" [KW13] and support our argument that practical aspects must not be neglected.

The data processing complexity is further increased based on required pre-processing techniques (e.g., digital filters) and the multitude of analyses that need to be performed. Leakage detection techniques such as Signal-to-Noise Ratio (SNR) [MOP07] or Test Vector Leakage Assessment (TVLA) [GJJ+11] have become popular due to their much lower processing complexity compared to key extraction techniques such as Correlation Power Analysis (CPA) [KJJ99; BCO04; CKN01; May00] or Mutual Information Analysis (MIA) [GBT+08; SMY09]. In addition, some algorithms can be performed in a non-profiled and profiled scenario with templates [CRR03; APS+06; GLP06; MOP07; SMS+17], further amplifying data storage requirements for different analyses and corresponding data acquisition patterns.

While leakage detection techniques may help avoid subsequent measurements by identifying leakage early, they are typically considered to be weak(er) distinguishers compared to key extraction techniques, i.e., actual leakage might be missed that a stronger statistical tool is able to extract [PGA+23]. Therefore, the best level of security is achieved by performing as many rigorous tests as possible, which however is limited by the available storage and processing time (and therefore, money).

In our work, we focus on these particular topics, to make SCA faster, while at the same time lowering storage requirements. Note how we avoid the term "computing" (implying a CPU-bound problem) and instead use the term "processing complexity". As backed by our experiments and perhaps contrary to commonly assumed wisdom in the side-channel community [BB17; RGV17], once properly optimized, many of the side-channel analysis problems are limited by file I/O and memory-bandwidth, as opposed to computational resources. We are unaware of previous works that view the implementation of SCA algorithms as a High-Performance Computing (HPC) optimization problem, where corresponding bottlenecks are identified and then resolved. For example, the Top-down Microarchitecture Analysis (TMA) method [Yas14; Int23] determines the utilization of pipeline slots in percent for the following categories: retiring instructions (more are better), slots affected by bad speculation (fewer are better), and slots that are front-end or back-end bound (fewer are better). Purely conceptual algorithm optimization that neglects microarchitecture properties of modern computing platforms may not unleash their full potential. Substantial fewer resources needed to perform a real-world side-channel analysis successfully can dramatically change the overall calculus of how secure an implementation is.

Based on our insight, we present our proof of concept implementation SCARR that is able to perform $1.7 - 2.3\times$ faster (Profile 1, Figure 5a) over uncompressed data compared to the thus far fastest framework, while performing $2\times$ to $15\times$ faster compared to the only other framework officially supporting file compression, while lowering the file size by up to 48%. Since the processing complexity is significantly bounded by I/O and also memory bandwidth on Intel platforms, using compressed data is beneficial, despite the computational burden of decompression prior to performing any computation. As a result, we are faster over compressed data than all other tested frameworks over uncompressed data (tested on Intel and Apple Silicon). In fact, on tested Intel platforms, processing compressed data is the fastest overall, while benefiting from reduced storage requirements!

1.1 Additional Motivation and Practical Relevance

SCA developed into a mature topic, with commercial offerings such as Inspector (Riscure), DPAWS (Rambus), Analyzr (Secure-IC), or esDynamic (eShard). There are also several frameworks (cf. Table 2) that are open-source but professionally developed out of semi-commercial interests, such as SCARED (eShard) [eSh19], LASCAR (Ledger) [LED18], and ChipWhisperer (NewAE) [OC14]. In our work, we focus on these robust frameworks only, as they have been actively maintained for more than 5 years, while acknowledging that other open-source frameworks exist (e.g., Daredevil [BB16; BB17], Jlsca [BK16], FOBOS [VK12], SCALib [CB23], SCA Toolbox [BKM+20], RamDPA [FHM+19]), some of which either have already been shown to be slower than SCARED[1] [Tim19; Bet23], or do not implement at least 3 out of 4 of our tested SCA algorithms[2] (SNR, TVLA, CPA, MIA), or do not support both int/float as input data (raw oscilloscope data or output from a digital filter), or do not support out-of-core processing (processed data larger than system memory), or lack advanced indexing for customized trace/point of interest selections to reflect more complex real-world attack scenarios.

We also want to emphasize the positive impact of the ChipWhisperer [OC14] ecosystem in the hardware security community, while pointing out (referencing their own documentation [DTO20]) that their toolchain is not optimized for speed, but instead, geared towards a good educational experience, which is why they recommend LASCAR and SCARED for performance-oriented analyses. While commending other framework authors for their professionalism, we would still like to discuss high-performance design patterns and how existing frameworks could be improved further. Moreover, we want to promote the adoption of Zarr as a file format for side-channel analysis. Zarr is a community-driven project aiming at efficient I/O for (distributed) parallel computing applications.

In terms of programming language and file formats, we note that other SCA frameworks and researchers predominantly use Python [eSh19; LED18; OC14] and HDF5 [HDF], also for popular datasets [BPS+20]. HDF5 (or `.h5`) is a fully-featured file format that enables combined storage of side-channel measurements and their metadata. HDF5 has several advantages over proprietary file formats such as `TRS`, for example, support for on-the-fly compression to reduce file size. At the same time, Python's HDF5 implementation `h5py` appears to suffer from several shortcomings (see Section 4) which further motivate our work. Considering the widespread adoption of HDF5 in a side-channel context, it is unsatisfying that the choice for HDF5 or any other file format was never analyzed or substantiated. Therefore, studying specifics of this file format and others, and their proper configuration, is warranted.

1.2 Contributions

- Study different file formats, their characteristics, and how they relate to side-channel analysis, with a strong focus on HDF5 and Zarr, but also covering NPY.
- Application-level analysis of high-performance patterns and file-handling techniques that align with one-pass statistical algorithms, out-of-core computations, and how performance changes based on I/O-oriented operations such as advanced indexing, batch-wise processing, chunking, compression, etc.

[1]Based on announcements from 2019 and 2023 with renewed claim that SCARED is "*best-in-class*" [Tim19; Bet23]. More in particular, in 2019 SCARED's CPA was shown to be faster than Daredevil, Jlsca, ChipWhisperer, LASCAR, and Riscure Inspector. We note that SCARED's performance is identical to its commercial variant esDynamic.

[2]Our brief inspection of these additional open-source frameworks did not indicate architectural or file format properties that would have warranted additional benchmarking within the scope of this work. We especially tested SCALib in combination with HDF5 files and could not reproduce previously claimed performance results that were based on synthetic, in-memory benchmarking.

- Microarchitecture-level performance analysis (Top-Down [Yas14; Int23] analysis) of side-channel analysis processing in different frameworks and corresponding tuning.
- Proposing SCARR, a Proof of Concept (PoC) side-channel analysis framework based on the Zarr [Zar15] file format that is able to outperform all other considered frameworks in our tests. SCARR is available on GITHUB (`https://github.com/hsrlab/scarr`) under open-source license and was submitted as CHES artifact.
- Comprehensive comparison of SCARED, LASCAR, and SCARR, showcasing that it can be beneficial performance-wise to use compressed data sets (Intel).

1.3 Outline

An overview of related work is provided in Section 2. Afterwards, in Section 3, we briefly introduce the necessary background of how modern computers work and their CPU microarchitecture. We continue by discussing different file formats and their conceptual differences in Section 4. The basic structure of SCARR and its configuration is then described in Section 5. This is followed by a Top-Down assessment in terms of utilization of the CPU microarchitecture, in addition to the benchmark-based comparison of the selected side-channel analysis frameworks in Section 6. Lessons learned and preferred design HPC patterns are summarized in Section 7. Afterwards, in Section 8, we briefly discuss limitations and bonuses of our proposed framework SCARR. Finally, conclusions are drawn in Section 9 based on the obtained results.

2 Related Work and Scope

Performance-oriented optimization of the side-channel analysis process can relate to improving the speed of the acquisition process or quality of the measurement data, e.g., as explained in [SM15; RWM19; BUS22], preference of leakage detection methods over key extraction methods [GJJ+11], selection of more optimal distinguishers that converge faster [HRG14; SMS+17; VS09], or optimizations of the algorithmic approach and minimizing arithmetic operations [LPR13; RGV17; BB17], while typically preferring one-pass algorithms that require file access only once [LPR13; SM15; SMG16; SMK+17]. Peak extraction and other pre-processing for Points-of-Interest (POI) selection are yet another option to possibly speed up the side-channel analysis, at the expense of increasing I/O complexity. Note, this concept is also called "trace compression" by some authors [BDG+14] and should not be confused with lossless file compression to reduce storage needs. In addition, our findings clearly show that Traces-of-Interest (TOI) and POI selection can incur a significant performance penalty due to increased I/O complexity.

It is interesting that the inter-dependency of Measurements-to-Disclosure and practical attack difficulty due to actual processing complexity was rarely discussed beforehand. Similarly, other metrics – while attempting proper theoretical modeling of the leakage – are not scaled based on, e.g., the time/energy resources to compute the desired metric. Consequently, a stronger theoretical attack might be outperformed by seemingly trivial practical efforts that are simply faster and require less time/energy, which is in essence the criticism captured in [KW13], which is why our work is geared towards practitioners.

Based on our findings given in Table 1, it is important to identify the processing-bounds, and the related foundational problem(s), as this can be a more effective way to boost evaluation performance (more traces processed in shorter amount of time), and therefore more accurately reflect the security of the assessed Device Under Test (DUT) in a real-world scenario. Consequently, we need to briefly discuss the technical scope, selection of algorithms, and the availability of the tested solutions (and their components).

Table 1: List of implemented side-channel analysis algorithms in SCARR, how they are bounded based on our experiments for one byte position using one-pass algorithms, including their foundational processing problem.

Algorithm	Processing Bounded	Foundational Problem
SNR	by file I/O; memory bandwidth	data partitioning
CPA	mainly by memory bandwidth	matrix multiplication
MIA	by compute (cores)	histogram building
TVLA	mainly by file I/O	basic statistics

Technical Scope. To enable a tailored discussion of high-performance design patterns, we need to address limitations of the Python programming language and corresponding packages, too. While our proposed concepts are generic in nature and could be implemented in any language, we discuss them within the context of Python only. Additionally, we focus on a "full-stack" benchmark considering real-world scenarios including TOI/POI selection (requiring advanced indexing, cf. Table 5), for data that is read from a disk. This is in contrast to similar efforts that exclusively focus on in-memory benchmarks over sequential data [Bet22; Cas22], therefore neglecting the complexities of I/O-oriented operations that can substantially diminish performance. Unless noted otherwise, we use the default configurations of the operating system and corresponding frameworks. Moreover, our work solely focuses on CPUs. GPU-implementations are considered *i*) out of scope and *ii*) are not included in the other tested frameworks either.

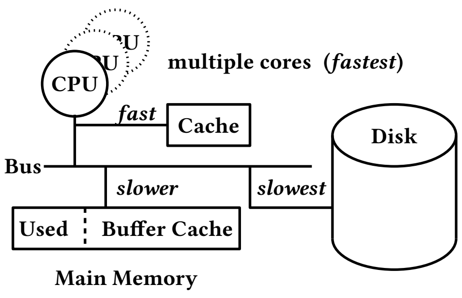
Selection of Algorithms. Our case study considers algorithms primarily designed for analyzing symmetric ciphers. We selected two leakage detection algorithms (SNR, TVLA) and two techniques for profiling and key extraction (CPA, MIA) as we consider them representative for a wide range of evaluation workloads. Each algorithm has its own foundational problem for processing, as listed in Table 1. To the best of our knowledge, there is no other work discussing the actual implementation of these algorithms under HPC considerations, and no previous attempt in making a determination of the bottlenecks that limit the processing of these algorithms (see Section 6). More specifically, whether they will be bound by the CPU Front-End (e.g., fetch and decode of instructions not fast enough), bound by bad speculation (e.g., branch mispredict), or bound by the Back-End (e.g., lack of memory bandwidth or compute cores). In other words: this approach can be applied to any other SCA algorithm and is not limited to the ones we selected.

Availability. We exclusively cover open-source frameworks and file formats as we cannot assess the performance of commercial solutions. We invite the respective companies to compare their software against SCARR using our representative benchmark-profiles (see Table 5). In addition, proprietary data formats such as ironArray were excluded (due to cost), despite claiming superior performance over HDF5 [HDF] and Zarr [Zar15]. The same is true for proprietary extensions such as ZIPVFS for SQLite.

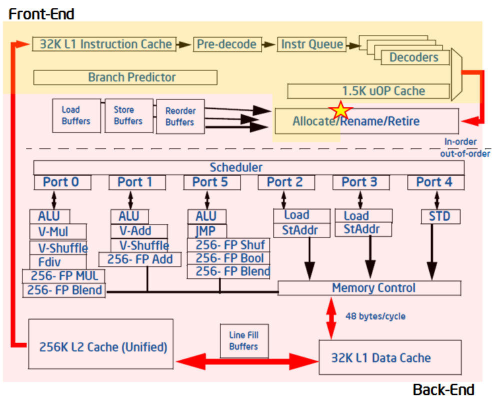
3 Computer Architecture System Model

In the following, we briefly recall basic properties of modern computer architectures [Dre07; Yas14; Int23]. As illustrated in Figure 1a, a representative system comprises a disk, main memory, cache, and multiple cores. In terms of disk performance, Operating Systems (OS) are designed to leverage various mechanisms such as read-ahead, and the buffer cache (residing in main memory) to optimize performance. Note that several resources are shared between the cores, and in terms of data throughput, they can outperform caches and any slower resource by orders of magnitude [Alt10], which is amplified by the shared resource problem.

Especially file retrieval from disk (or remote storage) is still considered slow, despite advancements such as NVMe and SSDs. This starvation of input data in the cores because of limited input data bandwidth can quickly lead to problems in the microarchitecture (cf. Figure 1b), as they can stall the pipeline. For optimum performance within a HPC-context, it is therefore necessary to actively use caches, and avoid other penalties such as branch mispredicts, that otherwise result in bad speculation [Int23] and diminished performance.



(a) Representative system architecture.



(b) Modern Intel microarchitecture [Int23].

Figure 1: System architecture model and Intel microarchitecture, illustrating various properties of modern systems, e.g., slow file retrieval, fast compute capabilities, multiple-dispatch of uOps (micro-ops) per cycle, and separation into front-end and back-end.

The microarchitecture of a modern Intel CPU can allocate four uOps (micro-ops) in its Front-End, while the Back-End can retire four uOps per cycle. This is used to derive the concept of a pipeline slot. Each slot represents the hardware resources needed to process one uOp. For a Top-Down analysis, the characterization assumes that for each CPU core, during each clock cycle, there are four pipeline slots available. A Performance-Measurement-Unit (PMU) then measures how well these pipeline slots were utilized. Through this process of classifying pipeline slots, it is possible to determine if a uOp was retired (completed successfully), or if it was mistakenly executed (bad speculation), or if the Back-End stalled it, due to being core- or memory-bound [Int23].

These four categories are also called a Level 1 Top-Down analysis. To read the PMU, we use Intel VTune. Note, the PMU offers more detailed counters to identify potential bottlenecks. As an important example, back-end bound can be further divided into core-bound (*simplified*: “not enough compute resources”) or memory-bound (*simplified*: “not enough memory bandwidth or too high latency”). For a tailored implementation of SCA algorithms, we consider it important to make such a determination to make targeted improvements. Note that this process is highly engineering-oriented due to the complexity of code, the OS, and the microarchitecture. Guidelines exist such as [Int23] to aid this iterative process but it is a problem that is difficult to solve fully analytically.

4 File Format Comparison

The choice of file format may appear as a seemingly trivial choice. Yet, different file formats exist and Table 3 lists various types that are used in the SCA community. Each format has been originally designed with a range of use cases in mind, with TRS being the only one specifically designed for SCA purposes. Even though literature exists on the topic of scientific data management [Rot09] including a synthetic comparison of Zarr and HDF5 [AB23], there is no specific comparison within the context of SCA. In the following,

Table 2: Overview of different side-channel analysis frameworks and some of their properties. At the time of testing, all frameworks were installed from GIT directly (approx. August 2023). The version number is therefore only a coarse-grained approximation of their code revision.

		File Handling			Parallel Processing			Dataset Type	
Framework	**Version**	**Format**	**Compression**	**Reader**	**I/O**	**Compute Parallelism**	**Process/Thread**	**Single**	**Tiled**
ChipWhisperer	5.7.0	NPY	not explicitly	synchronous	no	yes (numpy)	no / yes	yes	no
LASCAR	1.1	HDF5	not explicitly	synchronous	no	yes (numpy)	no / yes	yes	no
SCARED	1.1.0	HDF5	optional	synchronous	no	yes (numpy, numba)	no / yes	yes	no
SCARR	PoC	Zarr	optional	asynchronous	yes	yes (mp.pool, numpy, numba)	yes / yes	yes	yes

Table 3: Comparison of file formats commonly used in the side-channel analysis domain.

Format	Example	Compression	Chunked	Parallel Read	NumPy Types	Indexing	N-dimensional	Code Overhead
HDF5	[eSh19; LED18]	optional	optional[a]	not natively [b]	yes	yes	yes	none
TRS	[Ris18]	no	no	unknown	yes	yes (?)	no (?)	low
NPY	[OC14]	limited[c]	no	no	yes	yes	yes	none
SQLite	commercial	proprietary	no	yes	not natively	limited[d]	yes	substantial
Zarr	*this paper*	optional	yes[e]	yes[f]	yes	yes	yes	low

[a]Uncompressed datasets are contiguous by default. Compressed datasets are internally chunked (still appear as one file towards the OS).

[b]See h5py release notes for version 3.0: "*HDF5 has its own global lock, [releasing the GIL] won't speed up parallel data access using multithreading.*" Multi-processing is needed to work around this limitation while according to the documentation: "*avoid opening the file and then forking*".

[c]See text. Since no chunking is available, this is not an option for out-of-core computations.

[d]Trace indexing is fully supported by making appropriate SQL selections. Indexing of samples is natively supported for fields of type BLOB which is an array of uint8, but arrays of other numeric data types (e.g., int16) do not appear to be supported but we might be mistaken.

[e]Chunked files are stored as multi-file (sparse) storage when using the default Zarr DirectoryStore.

[f]From the Zarr documentation: "*Zarr will generally not block other Python threads from running*".

we provide a brief overview of file formats, their basic properties, and how they relate to SCA.

Initial Overview and TOI/POI Indexing. Each file format has specific characteristics that make it a better fit for a particular use case, and this makes choosing an appropriate file format an important task. As an example for very limited amounts of data, storing values in a Comma Separate Value (`CSV`) file might be acceptable. However, this creates a large storage overhead and performance penalty, as values are encoded as text, as opposed to binary. In addition, addressing (indexing) a specific value in a `CSV` becomes a challenge, as the length of the text-encoded values preceding it changes its position. Consequently, this file format is not used by side-channel practitioners.

Alternatively, storing time series data, such as recorded traces of a side-channel measurement campaign, by means of a contiguous binary concatenation is possible. Creating a `mmap()` (memory-map) of the file is then a convenient and fast way to read the data, which also benefits from OS-level optimizations through `madvise()` policies. This is the approach taken in Riscure's `TRS` file format, and NumPy's native `NPY` file format can also be memory-mapped. Due to its simplicity and high speed for sequentially read data, this is an intriguing choice.

At the same time, there are also disadvantages. Note that two-dimensional data sets (or higher dimensions) are stored as flattened arrays. As a result, accessing slices of N-dimensional arrays is only fast if the slices line up with the default structure of how they are stored. This can be a problem when indexing is needed, such as when selecting a time-range (slice, cf. Figure 2) within the measurement data and therefore, not reading the whole data set sequentially. Making such orthogonal selections over larger data sets directly will create many seek operations that can drastically diminish the read performance.

Potential alternatives are reading the data set in full and slicing it in memory, which is not possible for out-of-core computations where the data set size exceeds the available memory, and it creates a substantial read overhead if the needed slice is small relative to the whole data set. For complex real-world scenarios, the SCA analyst cannot know in advance which selection of points or traces may ultimately succeed. As a worst-case, multiple time-ranges need to be selected for testing, some traces need to be excluded if their alignment fails in the presence of countermeasures (TOI selection), etc. which turns this into a random access pattern that makes the reading slow if not counteracted at the file format level. Note how values are stored and processed in memory (C or Fortran order) will also greatly impact the performance based on the in-memory access pattern. Consequently, indexing is fully considered in our analysis profiles in Table 5.

Compression and Chunking. Another desirable feature is compression. Considering the widespread use of 12 bit ADCs in oscilloscopes and their benefit for power analysis, it is evident that when stored as `int16`, effectively 1/4 of the bits are overhead, independent of the potential compression ratio of the measurement data itself (which arguably may vary a lot). Still, the expected baseline for compressing the file size of such a data set is already 25%. A straight forward approach could be to manually compress the data set, to then decompress when needed to perform the side-channel analysis. This is a cumbersome process, possibly resulting in twice the needed amount of storage to temporarily store both data sets, and creates an unacceptable overhead when only requiring slices of the data. NumPy can automate this process for `NPY` files by storing/loading from `NPZ` – a zipped NumPy archive – into memory. However, this requires decompression in full to access any of the data, making this approach impractical. In addition, this again contradicts the idea of out-of-core computations with data sets exceeding available memory. Based on publicly available information, `TRS` does not appear to offer any type of built-in compression.

To overcome the inherent limitations of per-file compression of the whole data set, it is possible for some file formats to store data in smaller blocks called chunks. Chunks can be specified in an N-dimensional shape that reflect the most dominant access pattern.

The chunks themselves are again flattened, written to disk, and each chunk is indexed separately, e.g., by a B-tree in HDF5. When using compression with chunks, the chunks are self-contained blocks that can be accessed independently from others. Therefore, when dealing with larger data sets where only slices are needed, decompression is limited to the chunks actually required. This on-the-fly decompression when reading is supported by HDF5 and Zarr, but neither `NPY` nor `TRS`. Consequently, we focus our efforts on toolchains that support file formats that do support chunking (again excludes ChipWhisperer).

Determining the shape (and therefore the size) of a chunk must follow different considerations. For example, if chunking is based on a B-tree, then larger chunks for a given dataset reduce the size of the B-tree, making it faster to find and load chunks. At the same time, larger chunks can result in more overhead when only a small fraction of their data is needed. In addition, many small chunks may counteract the idea of using chunks, resulting in an excessive amount of seek and read operations, whereas too large chunks will exceed the L2/L3 cache of the CPU, making optimized decompression algorithms less efficient. For SCARR, the determination of the chunking configuration is presented in Section 5.

Aside from chunking, an optimized compressor must be selected, such as Blosc [The09] combined with lz4 [LZ411]. Blosc's stated goal is to provide faster transmission of data to the processor cache than the traditional, non-compressed, direct memory fetch. A standard fetch uses `memcpy()` to copy from memory. In contrast, by using the blocking technique [Alt10] in Blosc, the much higher speed of the CPU cache can be leveraged. By benefiting from shorter transfer times of compressed data, in-place decompression using `memmove()`, very high transfer speeds can be achieved that offset the additional compute load of the decompression. Note that lz4 and lz4hc (high compression), including their different compression levels (clevels), require the same time to decompress, making them an ideal choice for read-oriented applications. We consider side-channel analysis as a read-oriented application where data is recorded just once, and then analyzed many times over. Our benchmarks in Section 6 indeed confirm these benefits of using compression.

Parallel Reading. For some analyses, multiple byte positions may need to be studied at the same time. In addition, for EM side-channel campaigns, storing a tiled data set with all its XY-positions in a file is desirable to have it self-contained. This can result in situations where a time vs. memory trade-off in the corresponding SCA framework needs to be implemented. Either, more time is spent to (again) read from the same file when processing the next attacked byte position, or, more accumulators for a one-pass processing need to be available. Consequently, it may be desired to read from the same data set concurrently from different threads/processes to improve read throughput over time, and to scale more easily based on available core and memory resources.

Unfortunately, the standard HDF5 implementation has its own locking mechanism, which serializes a concurrent read access to a file. In contrast, e.g., using `mmap` and `fork` on Linux, it is possible to efficiently share a memory-mapped file in a Copy-On-Write (COW) fashion across multiple processes, with a minimum of resource overhead in read-only applications. As additional benefit, processes accessing the same data approx. at the same time, can retrieve the data from the buffer cache (as opposed to reading from disk again). Therefore, concurrent reads from the same file, or even parallel reads of the same data, can behave differently based on the chosen file format, and the internal process/thread structure of an SCA framework. See Table 3 for how different formats compare.

HDF5: Highlighted Properties. The Hierarchical Data Format (HDF) 5 [HDF] is designed to store large numerical arrays of homogeneous type, possibly organized in groups. This results in a file-system-like structure such as `file.h5/X/Y/traces` to store trace data recorded at various XY-positions of an EM campaign. Within the context of Python, `h5py` is the dominant library to access HDF5 files. As explained beforehand, indexing is routinely needed for TOI or POI selection, or to reduce compute and memory requirements. Unfortunately, the `h5py` documentation states [Dev23]: "*A subset of the*

NumPy fancy-indexing syntax is supported. Use this with caution, as the underlying HDF5 mechanisms may have different performance than you expect." in addition to: "*Very long lists [...] may produce poor performance*". Therefore, we were interested to test if there is any practical impact for side-channel analysts, which is why we investigate different indexing profiles (cf. Table 5), when comparing the different frameworks. Indeed, our results confirm unexpected behavior when indexing in HDF5-based frameworks is used (cf. Section 6).

Zarr: Highlighted Properties. Also aiming at the storage of large N-dimensional arrays, Zarr [Zar15] offers implementation details and features that are noticeably different compared to HDF5. As default, a so called `DirectoryStore` is used, that stores chunks of data as multi-file (sparse) storage in a directory. This can be used to implement a structure such as `directory.zarr/X/Y/traces` (cf. Figure 2). Since OS-level mechanisms are used for reading/writing, Zarr benefits from OS-level mechanisms which is in contrast to HDF5 where the library handles such aspects. Unlike HDF5, Zarr allows concurrent read or write access. Compared to default HDF5, a wider range of compression and filter mechanisms is supported, that can be more easily extended, too.

5 SCARR Overview and Configuration

In the following, we present a high-level overview of SCARR, our proof of concept SCA framework primarily written in Python. This section will only briefly outline the software architecture. A more complete documentation will be made available online. The same approach was taken, e.g., by the authors of ChipWhisperer [OC14].

SCARR uses Zarr as file format and aims at high-speed SCA processing over compressed data. As such, we need to focus on two aspects: *i*) how data is handled and *ii*) how results are computed. For a simplified view of the data handling, please see Figure 2. We follow a structure with EM tiles being groups, that hold traces and needed metadata as 2D arrays. Both uncompressed and compressed workflows are supported.

The framework is currently structured into a container class for computing, file handling, models, and engines to implement the needed base functionality for this work. Engines contain the implemented analysis algorithms with limited use of Numba and NumPy otherwise. The cryptographic algorithm, leakage model, etc. are modular and easy to extend, making SCARR comparable to other frameworks. All data processing follows the out-of-core computing principle and one-pass concept. Basic statistics needed use Welford's algorithm [Knu81] while the SCA algorithms themselves follow their respective publications. We refrain from presenting equations here, as later observed performance differences in SCARR were primarily achieved through basic microarchitecture analysis, corresponding optimizations as discussed in Section 6, and more efficient data handling and processing by means of tailored software architecture, as described in the following.

For processing a tile or multiple byte positions, separate processes are forked that may read concurrently from the data set. However, if the same data is read, we note that this would result in a buffered read through the buffer cache. To minimize run-time memory requirements and optimize read access, we retrieve multiple chunks in sequence to avoid excessive read operations. This unit is called a batch and ideally, a batch is asynchronously pre-fetched in parallel to the computation over the previous batch.

The optimum size of a batch depends on the type of the computation and platform parameters. Similarly, finding the best chunk parameters might not only depend on the intended access pattern, but also platform parameters. The Zarr documentation recommends that chunks should not be smaller than 1 MiB. While not specifically mentioned, when using compression, we assume that chunks should not be chosen too large to not exceed the size of the L2/L3 cache. In addition, too large chunks increase overhead when only requiring limited amounts of data from them.

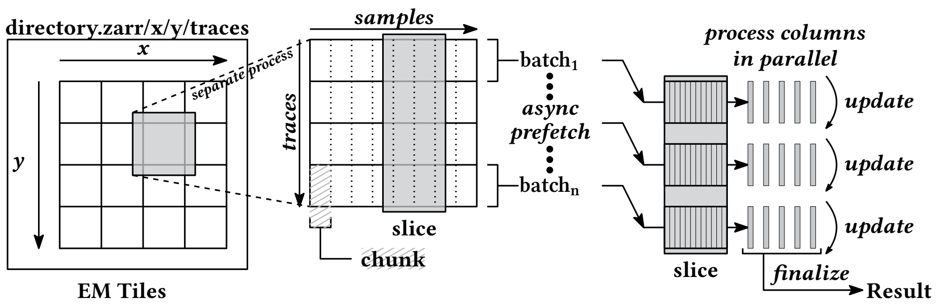


Figure 2: File and data processing structure in SCARR. Starting on the left: for each EM tile and byte position to be analyzed, a separate process is started. The data of each tile is then read in batches to extract the indexed slice. Each batch is fetched asynchronously to the parallel processing of columns of the batch that was fetched before. Each batch results in the update of internal accumulators to compute the desired result in a one-pass fashion.

We chose to empirically test a range of chunking configurations as illustrated in Figure 3a. For simplicity reasons, the batch size was tied to the first dimension of the chunking (equal to the number of traces being selected). From the results, it can be seen that (5000,1000) is a close-to-optimum choice on system Laptop (cf. Table 6), while avoiding too extreme parameters that might diminish performance on other platforms. We use the same chunking for both uncompressed and compressed data sets, i.e., unlike HDF5, our uncompressed data sets are also chunked.

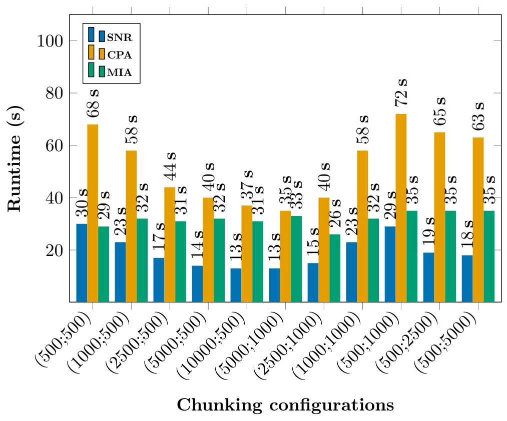
We want to emphasize that we are fully aware of benchmark challenges related to inadvertent buffering and caching, which is why we specifically tested the difference for all frameworks in Figure 3b. As expected, reads through the buffer cache are much faster than unbuffered reads from disk. Control over the buffer cache is possible, e.g., by using `vmtouch` to evict buffered pages. Since we assume one-pass processing, we specifically want to avoid benchmarking in the buffered scenario. Consequently, all other results presented in this work are unbuffered. In addition, we do not consider the CPU cache to be a problem, as the total amount of data processed exceeds the size of the CPU cache many times over. Our results in Figure 3b show that for limited-compute workloads such as SNR and TVLA, data fetching even from a fast SSD can still consume more than 50% of the processing time, with the actual computation being the other half.

When operating over `int` data, LASCAR's CPA does not offer a competitive run-time[9]. Therefore its corresponding data point was omitted in subsequent plots. Please be aware that these results are based on Profile 1 which is a sequential read over the data, an operation otherwise considered to be fast for storage systems.

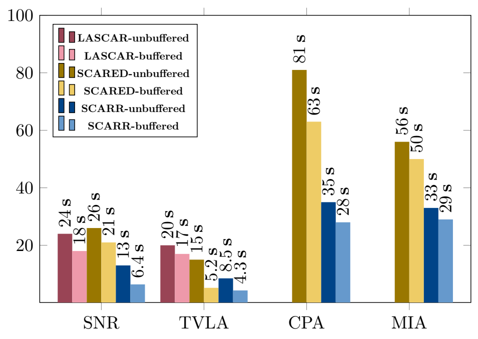
6 Full Case Study and Benchmarks

Prior to performing a Top-Down Level 1 analysis in Section 6.1, and presenting the full range of benchmarks in Section 6.2, we want to briefly discuss selected properties of our selected Profiles (Table 5) and the parameters of the analysis algorithms used. For creating the profiles, we followed practically relevant questions such as: are slices even worth selecting? Are strides worth it? In short: contiguous slices substantially reduce memory and compute requirements and the benefit is more significant the larger the discrepancy between trace and slice length. We caution against the use of non-contiguous lists as POI, as this might change the I/O behavior negatively with an increase in I/O effort

[9]The otherwise time-consuming step of a typecast to float is not done, preventing a more efficient use of OpenBLAS. Please note, LASCAR's batch size was manually set to 5000 for better results overall.



(a) Different chunking configurations of data set Medium, batch size in sync to the first chunk dimension and processing within Profile 1.



(b) Impact of buffer cache on run-time for all frameworks using Profile 1 (cf. Table 5) and data set Medium.

Figure 3: Determination of SCARR batch sizes and chunking, in addition to studying the impact of the buffer cache. Please note, all other benchmark results refer to the non-buffered scenario with data being read from disk. All results obtained on system Laptop (cf. Table 6). MIA is not implemented in LASCAR. See text for details.

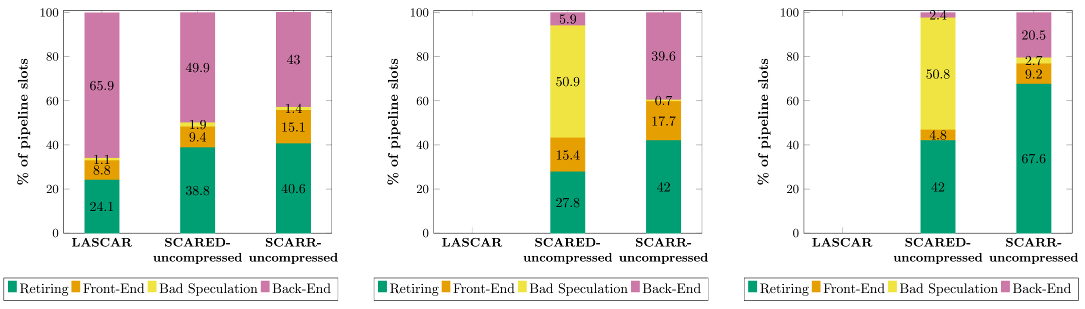
that outweighs the benefit of computing over less data. The same is true for strides that are equivalent to performing the same measurement but with lower sampling rate. For instance, for a 5 GS/s sampling rate, a stride of 5 will result in an effective sampling rate of 1 GS/s. Based on preliminary testing, the more complex indexing almost equalizes the gain of doing less computations, which is why we chose not to include it in any profile.

Regarding the analysis parameters: for SNR, we used 256 partitions, while for CPA, we perform a regular first-order attack using a Hamming Weight model targeting SubBytes. For MIA, we use the same and 9 bins, since targeting an 8-bit microcontroller (Medium data set). For TVLA, the combined data set has the dimensions as listed in Table 4.

6.1 Top-Down Level 1 Analysis of LASCAR, SCARED, and SCARR

A priority in software optimization is the identification of bottlenecks and resolving the dominating hotspots [Int23]. Therefore, we performed a Top-Down analysis of the three frameworks, for SNR, CPA, and MIA. The results of our analysis are shown in Figure 4. TVLA is excluded here for reasons of brevity. While no general statement can be made when a software is fully optimized, we refer to Intel's guideline that suggests a breakdown of pipeline slots as follows: 30-70% retiring, 20-40% backend-bound, 5-10% frontend-bound, 1-5% bad speculation (for optimized HPC applications) [Int23]. In addition, we emphasize that the *percentage* of pipeline slots, while indicative of the level of optimization, does *not* represent the *absolute* number of clockticks (CPU time), and therefore may not directly lead to a shorter run-time. All percentages given refer to overall pipeline slots available.

As can be seen for SNR in Figure 4a, the percentage of retiring instructions gradually increases from LASCAR (24.1%), over to SCARED (38.8%), and SCARR (40.6%), while being back-end bound gradually decreases from 65.9% (LASCAR) to 43% (SCARR). Looking further into the details of the back-end bound category, we reveal that, e.g., SCARED is primarily core-bound with 38%, with only 11% being memory-bound. In contrast, SCARR is much less core-bound with only 25.7%, while increasing the memory-bound to 16.6%. LASCAR shows the lowest core-bound of 23.3% and is 42.6% memory-bound that is architecturally attributed to lack of memory bandwidth, and code-wise



(a) TMA of SNR (Profile 1) **(b)** TMA of CPA (Profile 1) **(c)** TMA of MIA (Profile 1)

Figure 4: Results of the Top-down Microarchitecture Analysis (TMA) method of the three frameworks LASCAR, SCARED, and SCARR. Unfortunately, a Level 1 Top-Down analysis could not be performed for LASCAR's CPA due to excessive runtime. MIA is not implemented in LASCAR. As later seen in Section 6, a higher percentage of retiring instructions is indicative of a better performance.

likely stem from unneeded memory copies. Especially for SNR, we consider the results of all frameworks as quite unoptimized but our priority was CPA and MIA, as they are more demanding workloads. Note that the effective physical core utilization for LASCAR (15.4%), and SCARR (19.1%) are quite low, clearly indicating that SNR is not a "compute" problem, while SCARED shows a utilization of 55.4%. We want to point out that the implementation of the SNR partitioning is challenging, as it cannot be easily vectorized in a Single Instruction Multiple Data (SIMD) manner. Processors would need to perform a costly gather from memory and perform a scatter store, all of which are expensive operations. Since memory locations are only accessed once, applying blocking is of practically no use. Due to these inefficiencies when implementing SNR, we do not consider it a particular good choice in terms of return on energy investment.

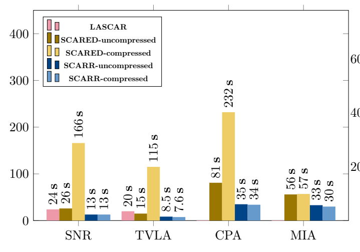
Moving on to CPA in Figure 4b we note that a Top-Down analysis could not be performed for LASCAR due to excessive run-time. For SCARED, it is striking that half of the pipeline slots are wasted due to bad speculation (50.9%), of which 35.4% are branch mispredicts and 15.5% machine clears. Mispredicts are typically the result of un-optimized if/else branching that are especially costly with load/store operations from/to memory. In contrast, SCARR has a negligible amount of bad speculation (0.7%). Since CPA is in essence the implementation of a matrix multiplication which is an inherently memory-bound problem, seeing a 39.6% back-end bound in SCARR is not surprising, with an almost equal split between memory-bound (19.4%) and core-bound (20.2%). SCARED is only 5.9% backend-bound, due to the substantial losses as a result from bad speculation, indicating needed optimizations. In this case, CPU utilization between the frameworks is comparable at 47.5% (SCARR) and 42.9% (SCARED).

We want to conclude this Top-Down analysis by inspecting MIA, as shown in Figure 4c. MIA is not implemented in LASCAR and was excluded from this analysis. Again, we observe high bad speculation (50.8%) in SCARED that is the result of branch mispredicts (48.0%) which can be traced down to their histogram implementation. In contrast, there is negligible bad speculation in SCARR, resulting in many more retiring (67.6%) pipeline slots compared to SCARED (42%). SCARED shows an effective core utilization of 54.1% vs. 83.8% in SCARR. We consider SCARR's MIA quite optimized and in-line with the HPC guidance from Intel, despite using "just" Python with a limited amount of Numba-generated code. As can be seen in the next section, indeed, better results from the Top-Down analysis tend to translate into less run-time overall.

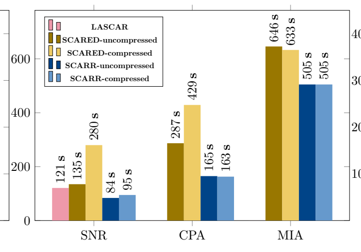
6.2 Benchmark Comparison of LASCAR, SCARED, and SCARR

After the initial Top-Down analysis, we put all frameworks to a test by analyzing data set Medium (Table 4) and applying the analysis profiles of Table 5. These profiles were chosen to emulate different real-world scenarios. This data set for the bulk of our benchmarks is the result of a power analysis using the ChipWhisperer Husky, recording all rounds of a software AES-128 running on an XMEGA target.

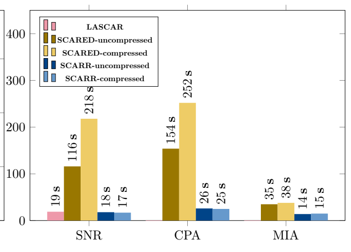
For compressed data sets for SCARED, we used their `compress_ets()` to convert to compressed HDF5. Please note, the implementation of this function performs unneeded compression of all data, including plaintext (chosen at random), ciphertext (random), and key. In contrast, such metadata remains uncompressed in SCARR. We rounded all run-times of more than 10s, while results with less than 10s include the first decimal position. All benchmark results from Figure 5 are discussed hereafter.



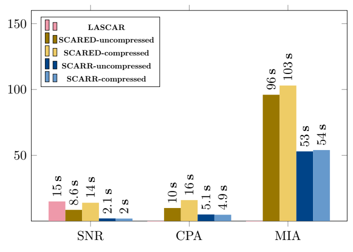
(a) Profile 1 – Laptop



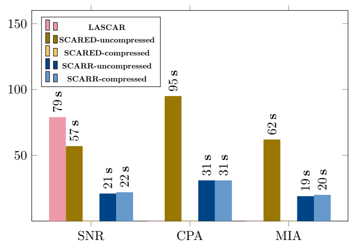
(b) Profile 2 – Laptop



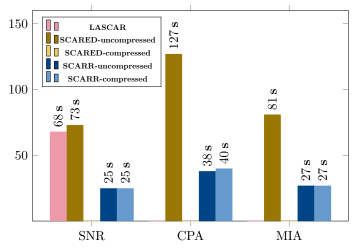
(c) Profile 3 – Laptop



(d) Profile 4 – Laptop



(e) Profile 5 – Laptop



(f) Profile 6 – Laptop

Figure 5: Benchmarking of side-channel analysis frameworks on a Laptop (cf. Table 6), based on the Profiles in Table 5, over data set Medium (cf. Table 4).

Profile 1 This profile represents the standard attack scenario of analyzing one byte position of the desired algorithm (SNR, CPA, MIA), or distinguishing two sets of traces (TVLA), while reading the whole data set. To somewhat align the results in our plots and avoid excessive run-times for benchmarking, we chose to limit MIA to 5000 sample points. The resulting plot in Figure 5a shows the performance of all three frameworks, including the analysis over compressed data. SCARR outperforms all other frameworks by approx. a factor of two which is true even when comparing compressed data sets against other frameworks operating over uncompressed data. LASCAR provides decent performance for SNR and TVLA, but was excluded from CPA due to excessive run-time of approx. 30 minutes which would have prevented a reasonable illustration in our plots. SCARED offers an average performance for uncompressed data, but is affected by a dramatic increase in processing time when working over compressed data. Possibly, this is the result of a less than optimal choice of compression parameters, or otherwise performance-limiting aspects in their file reader. Please note that for MIA, only 5000 samples points are processed, therefore, minimizing the compression penalty otherwise observed in SCARED.

Profile 2 This profile represents the scenario when the recipe for extraction is already known, and the extraction of all 16 byte positions is needed, while again using all of the available data for the analysis, resulting in a comfortable sequential read over the data.

Note that we excluded TVLA from this profile as it is not applicable within this profile.

For operating on many byte positions, LASCAR is able to outperform SCARED for SNR, presumably due to the much lower CPU utilization observed during Top-Down analysis. Here, the advantage of SCARR is again comfortably outside any misinterpretation. Aside from SNR, compression again only has insignificant impact in SCARR (if at all). For SCARED, presumably due to how I/O is handled and cached, relative loss in performance for compressed data is not as significant as for Profile 1 but still noticeable.

Profile 3 In this profile, we demonstrate the negative performance impact TOI indexing can have. In particular SCARED is affected by TOI indexing, while LASCAR and SCARR show performance that is similar to Profile 1. SCARED shows a performance that is much worse compared to Profile 1, despite computing only over half of the traces. This can only be explained by a much worse I/O behavior of their file handling, as LASCAR uses HDF5, too. LASCAR performs similarly to SCARR, while compressed data in SCARR is still marginally faster than LASCAR. Again, we see a massive drop in performance over compressed data in SCARED (SNR, CPA), while again pointing out the restricted number of samples for MIA which limits this effect.

Profile 4 As recorded traces can be overly long, analysts typically choose a range for their SCA evaluation of the data. For this scenario, we chose to use all traces available, while limiting the sample range to the same length for all algorithms (SNR, CPA, MIA), offering a fair comparison in terms of run-times. Since the range is approx. 1/8th of the overall sample range, a reduction of processing time by a factor of 8 might be expected. Unfortunately, this is not the case for any of the studied frameworks, as the increased complexity of indexing does not result in a linear decrease of the processing time. As can be seen from the results, all frameworks handle the selection of time ranges fairly well relative to their results of Profile 1.

Profile 5 and 6 In the presence of strong countermeasures for complex targets, one of the pre-processing steps prior to the actual analysis may include the alignment or selection of traces of interest. Consequently, not all traces can be used, either because certain patterns are not found, or their alignment fails. As result of the pre-processing, we assume that a virtual index is created over the original data set, which then is used in subsequent attempts to analyze the data or to store it separately. To emulate creation of such a virtual index, we randomly generated an ordered index of traces (based on the same seed for repeatability) to then select traces of interest from the data set. For Profile 5, we assume that 60% of the traces can be used, while for Profile 6 this number is 80%.

This advanced index was then passed to the frameworks. In case of LASCAR, this causes data to be read in full prior to any computation, as passing an advanced index to HDF5 causes it to return values (as opposed to providing a view), which is a known `h5py` limitation, thereby violating the out-of-core requirement. SCARED does work over such a virtual index and achieves reasonable performance over uncompressed data. However, the same index over compressed data causes SCARED to show an excessive run-time (>1h) which is why we terminated its process.

Summary. In all cases, SCARR performs similar to Profile 1, while showing somewhat of a performance degradation for SNR, as data cannot be fetched quickly enough by the asynchronous file reader. Neglecting this minor deficiency, this confirms our overall design rationale for SCARR and its formidable performance over compressed data. For the sake of completeness and to check behavior under less favorable conditions, we did limited testing over data set Large to confirm the observed behavior also for this simulated data set. While the results were fully consistent for the uncompressed scenario, we lost the benefits of compression due to fully using 16-bit with high levels of simulated noise, resulting in a much lower compression ratio, while still performing faster than the other frameworks.

Table 4: Different data sets for benchmarking and comparison of compression levels. Due to the vast amount of processing, we focus on the data set Medium for benchmarking. Small data set is ASCAD [BPS+20]. Tiled data set was not converted to HDF5, as none of the other frameworks support it. Medium and Tiled will be made available under `https://github.com/hsrlab/scarr` and also be part of our artifact submission.

Dataset	SCA Properties		Data Set Properties				Size in GiB	Compressed in GiB [reduced by %] (chunking)	
Name	Algorithm	Measurement	#tiles	#traces	#samples	type	uncompressed	HDF5[a]	Zarr[b]
Small	(SW)AES	Power (8-bit)	1	60k	100k	int8	5.6	2.9 [-48.2%] (235,782)	3.2 [-42.9%] (5000,1000)
Medium	(SW)AES	Power (12-bit)	1	100k	70k	int16	14.0	9.6 [-31.4%] (391,547)	7.2 [-48.6] (5000,1000)
Large	AES	Simulated (16-bit)	1	10M	5k	int16	96	not assessed	92 [-4.1%] (5000,1000)
Tiled	(HW)AES	EM (8-bit)	15,8	20k	20k	uint8	46	not assessed	33 [-28.3%] (5000,1000)

[a]Compression parameters: DEFLATE/ZIP is the default in HDF5 and also SCARED (with compression level 9). Metadata is compressed/chunked, too.
[b]Compression parameters: lz4hc with compression level 9 and Blosc.shuffle. Metadata remains uncompressed and is chunked as (5000,16).

Table 5: Benchmark profiles for different analysis scenarios (see text), covering varying number of byte positions to attack, Trace-of-Interest (TOI) selection, and sample Points-of-Interest (POI) index. Due to excessive run-time, we chose to run MIA over a shorter range of samples only.

			Samples Index (POI)		
Profile	Byte Positions	Trace Index (TOI)[a]	SNR/TVLA/CPA	MIA	Note
P1	1	[::1] (all)	[::1] (all)	[0:5000:1]	Sequential read of all traces (use whole data set).
P2	16	[::1] (all)	[::1] (all)	[0:5000:1]	Attacking all byte positions while using whole data set.
P3	1	[::2] (every 2nd)	[::1] (all)	[0:5000:1]	Only working on every 2nd trace (structured TOI selection).
P4	1	[::1] (all)	[48739:57413:1]	[0:8674:1]	Same POI length for direct run-time comparison.
P5	1	random (60% of traces)	[::1] (all)	[0:5000:1]	Randomness from seeded pseudo random number generator.
P6	1	random (80% of traces)	[::1] (all)	[0:5000:1]	Randomness from seeded pseudo random number generator.

[a]The index notation follows NumPy's syntax of [start:stop:stepsize].

Table 6: System configurations for benchmarks. Laptop and Server running Ubuntu 22 LTS with Kernel 6.2, Python 3.10.12, and NumPy 1.23.5. Apple with MacOS Ventura, Python 3.11.4, and NumPy 1.24.4. Apple and Intel both with OpenBLAS as NumPy backend. Intel platforms are with Hyperthreading. The given memory bandwidth is the theoretical system bandwidth when using all channels. Linux storage formatted as ext4.

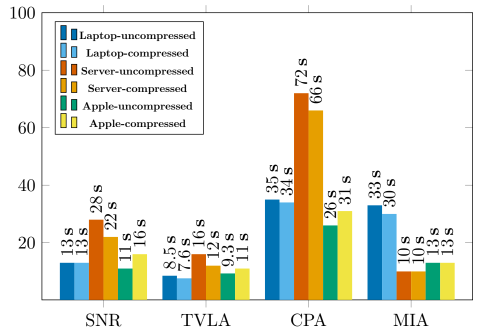
System	CPU (Base Frequency)	Node	#CPUs / #Cores / #Threads	Memory (Type, Bandwidth)	PCIe	SSD Storage (Type)
Laptop	Core i7-11700H (2.3 GHz)	14 nm	1 / 8 / 16 (16 total)	64 GB (DDR4-3200, $2 \cdot 25.6$ GB/s)	4.0	NVMe (Samsung 980 Pro)
Server	Xeon 8276M (2.2 GHz)	14 nm	$4 / 4 \cdot 28 / 4 \cdot 56$ (224 total)	6.5 TB (DDR4-2666, $6 \cdot 21.3$ GB/s)	3.0	NVMe (Samsung 980 Pro)
Apple	M1 Pro (3.2 GHz)	5 nm	1 / 8P + 2E / 8P + 2E (10 total)	16 GB (DDR5-6400, $4 \cdot 51.2$ GB/s)	4.0	NVMe (Apple)

6.3 Comparison Across System Architectures

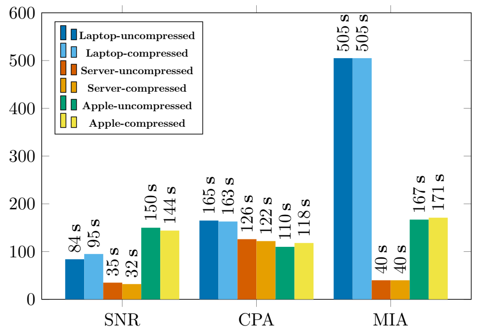
In the following, we focus on the performance of three systems with different platform architectures, as listed in Table 6. The purpose of this comparison is to illustrate the behavior when offering different number of cores (compute-capability) and memory-bandwidth and latency (memory-capability). We are comparing three systems: Laptop, Server, and Apple. Laptop and Server are the same technology node with similar microarchitecture and AVX512. In addition, they almost have the same base frequency. However, their number of cores is fundamentally different, as Server is a multi-socket system with a total of 224 threads, while Laptop only has 16 threads. Compared to Server, the memory bandwidth per channel is higher on Laptop due to higher frequency, while Server has more channels. Laptop uses low-latency memory. File I/O throughput is limited by PCIe3 on Server compared to PCIe4 on Laptop. At first sight, Server offers much higher compute capabilities while being limited in terms of I/O and memory throughput.

In contrast, system Apple contains an M1 Pro with the least number of threads total (and limited overall memory of 16 GiB), while offering by far the highest memory bandwidth per channel and overall system bandwidth. Practical I/O speeds over PCIe4 are assumed to be similar to system Laptop but were not assessed in a stand-alone benchmark.

Running SCARR with Profile 1 and Profile 2 (cf. Table 5) over the Medium data set results in Figure 6. Looking at Figure 6a, we can see that for SNR, system Apple performs similarly to Laptop. Server is by far the slowest which appears surprising but can be explained as follows: SNR is a memory-bound problem (both bandwidth and latency), and file I/O on the server is slower due to PCIe3, resulting in cores to be starved of data. To make this situation worse, neither SCARR nor any of the other frameworks are optimized for a Non-Uniform Memory Architecture (NUMA). Server is a four-socket crossbar NUMA architecture and we are creating Numba-generated compute threads without specific CPU affinity, resulting in remote memory access, dramatically increasing memory latency. Indeed, Intel VTune reports 100% remote memory access utilization through one CPU, while the core utilization in any other CPU remains close to 0.1% – despite "computing" the SNR. In short: this problem severely amplifies the already limited memory bandwidth and latency situation in Server.



(a) SCARR System Comparison – Profile 1



(b) SCARR System Comparison – Profile 2

Figure 6: SCARR processing of data set Medium (cf. Table 4) on different systems, using Profile 1 and 2 (cf. Table 5). See text for explanations.

Note, even if the software were rewritten to follow a cluster-oriented processing-style of map-reduce, with parallel readers for each NUMA node, thereby avoiding remote memory access, it is still likely to be limited by PCIe3 for I/O. Alternatively, compressed data can be used to relieve pressure on the memory bus. As a result, Server shows the most substantial gain by using compressed data, as the disparity of compute-power relative to data-throughput is the most significant among the compared systems.

For TVLA in Figure 6a, we again see that Server is the slowest, which again is explained by lower I/O speed and memory. Just as before, using compression helps to improve performance. Laptop and Apple perform in a similar way, with minor differences presumably attributed to different SSD models, as the core utilization remains low.

For CPA in Figure 6a, again, Server is the slowest but this time, the reasons are slightly different. Due to the fact that Server is a NUMA architecture, and Python is a single process constrained by the Global Interpreter Lock (GIL), threads created by the NumPy backend OpenBLAS are confined to one CPU. Therefore, the performance of Server is that of one of its CPUs while all other CPUs remain unused. Since CPA is essentially a matrix multiplication problem for computing the correlation, it is inherently memory-bound. Furthermore, the processed data chunks are rather small, thereby likely preventing any gains from multiple memory channels. As a result, the server performance is negatively impacted by the lack of memory-bandwidth and high latency. Note that these limitations exist in the other frameworks and also commercial software such as esDynamic, too.

Unlike the problems before, MIA in Figure 6a represents a core-bound problem. In addition, Profile 1 operates only over a limited amount of data, drastically minimizing the I/O burden. As can be seen, Server is the fastest by a small margin, ultimately benefiting from its many cores. Due to the limited amount of data being processed, compression only has a negligible effect on the overall processing time.

For Figure 6b, we are considering Profile 2. It is striking that Server performs much better across all benchmarks compared to Profile 1. Since the analysis over different byte positions relies on the same data, which is then buffered (and possibly cached) when accessed multiple times, Server is no longer limited by I/O or memory bandwidth and latency, such that all available cores can be leveraged. The increase in processing time is especially minor for SNR and MIA, while more noticeable for CPA. Note, for CPA, we experimented also with NUMA optimized code, reducing the processing time by about 15s, but we chose to not include this to ensure consistency of the code being used.

Other systems show a mostly balanced increase in processing time that showcase the complex interaction between microarchitecture utilization, number of cores, memory-bandwidth and latency, and I/O. While in most cases, compression still is beneficial, on Laptop, we observe a little bit of a slow-down for SNR. Unfortunately, we could not determine the root cause of this. Looking at the MIA results of Server, it is impressive how the processing time of 40s is about 12.6x less than that of Laptop, which resembles the factor by which their number of cores differ (224 threads vs. 16 thread = 14x), while accounting for some overhead and slower memory. Based on these results, it is evident that substantial differences across different system architectures exist. While the overall behavior did not change for the limited set of systems we tested (SCARR was fastest), clearly, different scaling effects are observed in the different frameworks. Due to the limited added value of these results and lack of space, we chose to not include them.

7 Lessons Learned – High-Performance Design Patterns

Maximizing the performance of existing hardware, or planning ahead for the next purchase of computer hardware can greatly impact evaluation performance. This is only possible if informed decisions can be made. Therefore, we present our lessons learned throughout the process of writing an optimized SCA framework.

DP1 – Identify bottlenecks. Our Top-Down analysis revealed unexpected deficiencies in other frameworks that indeed do affect performance. Awareness and moderate tuning of the code can already greatly improve performance. Knowing about current software bottlenecks and microarchitecture resource utilization can help ensure to purchase more capable hardware to mitigate previously existing architectural limits.

DP2 – Avoid conditional branching especially with memory write-backs.

Computation of SCA algorithms may require conditional branching, which causes the CPU to speculate about the outcome of the condition. Just like in GPUs should branching be avoided if possible. For example, nested if/else conditions in a for-loop will cause massive bad speculation, which is what we observed in other code.

DP3 – Increase variable locality. To benefit from compiler optimizations and avoid direct memory write-backs, improving variable locality can help to benefit from caching. While this technique may appear contrary to initial optimization goals, as it results in more code and variables, the benefits clearly outweigh solely code-oriented quality criteria.

DP4 – Do not make one-pass processing a dogma. While one-pass processing has become the de-facto standard for processing side-channel data, we do not consider it a strict principle, for as long as a multiple-access pattern is restricted to the same chunks that can be buffered, such that multiple reads can be retrieved from the buffer cache or CPU cache. This can help to more easily scale parts of the software architecture.

DP5 – Avoid unneeded memory copies. In Python, it can be tricky to avoid unneeded memory copies. Especially for algorithms that tend to be memory-bound already, such as SNR, or CPA, creating such memory copies will create additional back-pressure on the memory bus. Taking into account that memory is slower than the CPU (or its cache), this can cause an additional undesirable penalty for effective pipeline utilization.

DP6 – Hide disk latency by asyncing. Once computation is sufficiently demanding and is more burdening than I/O, data should be asynchronously prefetched, especially when performing compression or more complex indexing. Otherwise, fetching data serializes the processing due to data not arriving in the CPU fast enough, or due to the GIL preventing other threads from being executed in the meantime.

DP7 – Use caching and compression where appropriate. Data throughput into the CPU can be quite limited. Loop blocking and tiling are standard techniques to benefit from caching, which helps to overcome memory bandwidth limits. Additional gains are possible by compressing the data, for as long as this is offset by sufficient compute power. This is also a desirable option for offloading data onto a GPU.

8 Limitations and Bonuses of SCARR

Limitation: Limited range of algorithms. Our proof of concept was a time- and resource-limited effort (cf. Acknowledgements) to make the case for compressed data sets and raise awareness for potential improvements in some of the other frameworks. As such, the overall range of (pre-) processing algorithms, analysis options, etc. is limited and SCARR is currently not a drop-in replacement for commercial frameworks.

Limitation: Zarr still being quite slow. Additionally, since `Zarr-Python` was mainly developed for file retrieval from remote storage, some internals are not fully optimized for our use-case. As backed by our experiments, Zarr (and HDF5) slow down the processing substantially. For example, the TVLA ratio of CPU time is approx. 1:2 between I/O and compute, but with substantial gaps between computations as this CPU time is unevenly split between 2 threads for reading and many threads for the computation. This also makes our asynchronous pre-fetch fail. We are currently in touch with Zarr developers to optimize this behavior for our low-latency, one-pass, out-of-core oriented processing, possibly aiming at a Zarr implementation leveraging `io_uring` in a native language.

Limitation: Numerical accuracy not assessed. Furthermore, we optimized for speed while trying to avoid algorithms and functions that could possibly result in numerical instability. We did not specifically assess the numerical accuracy in our framework, or any other. For example, based on our opinion, the SNR computation of the noise component in SCARED is based on a naive, one-pass variance approach that is prone to catastrophic cancellation. Data type precision and more intricate problems of, e.g., differing precision along different axes when using, e.g., `np.sum`, were not the primary focus of this work.

Limitation: Uniform configuration. We tried to make this a fair comparison by selecting only one set of uniform parameters that is static across all systems, despite knowing that per-system tweaking of SCARR would give better results. In contrast, SCARED does internal and automatic tuning of, e.g., the batch sizes based on data set dimensions. In the future, we may consider such optimizations, too.

Bonus: Tiled data sets. As evident from our description, we are the only framework to directly support EM tiles in a way that is also very R&D-friendly. For example, once suitable EM tiles are identified for a more detailed analysis, other sub-directories (corresponding to unneeded tiles) can be removed from the data set simply by removing the directories with OS-level mechanisms, thereby avoiding 3rd party tools or scripting.

Bonus: Unlimited storage on Box.com (possibly others). Bulk data storage providers such as `Box.com` only theoretically offer unlimited storage. Quite often, the size of single files is limited to, e.g., 50 GiB (enterprise account), while the overall storage indeed can be up to 5 PiB. File formats without sparse storage cannot be used in combination with such services. In contrast, Zarr's DirectoryStore with properly sized chunks enables a convenient way to work around the per-file limitations of these services.

Bonus: Cloud-native format. Zarr was developed with cloud-native capabilities in mind. As such, it can be used in combination with cloud-storage systems such as S3 [Ama06] or N5 [Saa17] easily. Please note that the high-latency of file retrieval may require a different configuration compared to what we presented in this paper.

9 Conclusion

By following high-performance design patterns and selecting Zarr as file format, we could demonstrate that side-channel leakage detection metrics are a substantially I/O-bound problem (SNR, TVLA), while the computation of a cache-optimized MIA becomes primarily core-bound, with moderate I/O improvements by asynchronously pre-fetching data. CPA is in between these workloads and benefits significantly from increased memory-bandwidth, as otherwise the cores are starved of data too quickly (Intel). This problem may get worse in the future with new CPU instructions being released to the market by both Apple and Intel (the latter currently not supporting float32/64 though), specifically aiming at faster matrix multiplication (AMX). To maximize performance within a Python framework, we also showed how different parameters such as chunking and batch size are chosen.

By way of example, we then demonstrated that it is possible to achieve noticeable performance gains over more mature frameworks. Our proposed software architecture is vastly superior and creates a win-win situation: compressed data sets are likely to take up less space on disk, while at the same time, they can speed up the processing, as the I/O burden gets less since fewer data must be read (Intel). While only tested on a limited number of platforms, we are convinced that the selection of systems is representative for a wider range of platforms. We could also demonstrate that if memory-capabilities are not limiting but instead cores (Apple), indeed decompression results in a marginal performance loss, while pointing out that Apple was not the platform for which we tuned our code.

Due to the diversity in compute platforms (including NUMA systems), and the need for proper configuration, it would be unreasonable to make claims towards having the fastest framework without more thorough testing. Also, there will be artificial corner cases where our static configuration will deliver sub-optimal results. In addition, due to how we support tiled data sets (through mp.pool) incurs a set up cost which cannot be avoided when operating over few points only. Implementations with more simplistic (contiguous) file formats and access patterns, and higher degree of native code may perform faster over sequentially read data. Therefore, we only claim to (temporarily) have the fastest framework over compressed data within the technical scope described in Section 2. However, we are fully confident that SCARR performs very well for all real-world scenarios.

While Zarr performed in a more respectable way than HDF5, we emphasize that both file formats slow down limited-compute workloads such as SNR and TVLA, especially when performing any type of indexing. In the future, we expect further performance improvements by Zarr Version 3, as it introduces new features such as "Shards" that could benefit SCA-oriented workloads. Further gains also could be achieved by implementing a custom bit-packing shuffle filter for compression, that takes the specifics of unused bits in an integer value more directly into account by masking them out.

Exciting new hardware currently being released includes PCIe5, Intel Xeon Max CPUs with HBM2 memory, and Apple Pro/Max/Ultra CPUs with a memory bandwidth up to 800 GB/s! These developments may further change the overall calculus of performing side-channel analysis at scale, and how algorithms should be implemented for optimum performance. We hope by providing many pointers for potential optimizations, industry practitioners are inspired to improve their frameworks to more accurately reflect the needed effort to attack real-world cryptographic implementations.

Acknowledgments

Jonah Bosland has been funded through the Office of Naval Research (ONR) Workforce Development grant (N000142112552) during June-November 2023, as part of his undergraduate studies. Stefan Ene has been funded through the Summer 2023 Research Experience for Undergraduates (REU) program by the School of EECS at Oregon State University.

References

[AAR+03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side—Channel(s). In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, Lecture Notes in Computer Science, pages 29–45, Berlin, Heidelberg. Springer, 2003.

[Alt10] Francesc Alted. Why Modern CPUs Are Starving and What Can Be Done about It. *Computing in Science & Engineering*, 12(2):68–71, March 2010. URL: http://ieeexplore.ieee.org/document/5432301/.

[Ama06] Amazon. Cloud Object Storage – Amazon S3 – Amazon Web Services, Amazon Web Services, Inc., 2006. URL: https://aws.amazon.com/s3/.

[AB23] Sriniket Ambatipudi and Suren Byna. A comparison of HDF5, zarr, and netCDF4 in performing common I/O operations, 2023.

[APS+06] C. Archambeau, E. Peeters, F. -X. Standaert, and J. -J. Quisquater. Template Attacks in Principal Subspaces. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, Lecture Notes in Computer Science, pages 1–14, Berlin, Heidelberg. Springer, 2006.

[ABB+20] Melissa Azouaoui, Davide Bellizia, Ileana Buhan, Nicolas Debande, Sèbastien Duval, Christophe Giraud, Èliane Jaulmes, François Koeune, Elisabeth Oswald, François-Xavier Standaert, and Carolyn Whitnall. A Systematic Appraisal of Side Channel Evaluation Strategies. In Thyla van der Merwe, Chris Mitchell, and Maryam Mehrnezhad, editors, *Security Standardisation Research*, Lecture Notes in Computer Science, pages 46–66, Cham. Springer International Publishing, 2020.

[BUS22] Davide Bellizia, Balazs Udvarhelyi, and François-Xavier Standaert. Towards a Better Understanding of Side-Channel Analysis Measurements Setups. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science, pages 64–79, Cham. Springer International Publishing, 2022.

[BPS+20] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *Journal of Cryptographic Engineering*, 10(2):163–188, June 1, 2020. URL: https://doi.org/10.1007/s13389-019-00220-8.

[Bet22] Guillaume Bethouart. Benchmarking Side-Channel solutions... Why? How? April 7, 2022. URL: https://eshard.com/posts/benchmarking-side-channel-solutions.

[Bet23] Guillaume Bethouart. 'Scared': Better, Faster, Stronger. June 27, 2023. URL: https://eshard.com/posts/scared-side-channel-attack-library-enhanced.

[BDG+14] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. Side-channel leakage and trace compression using normalized inter-class variance. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '14, pages 1–9, New York, NY, USA. Association for Computing Machinery, June 15, 2014. URL: https://doi.org/10.1145/2611765.2611772.

[BB16] Paul Bottinelli and Joppe W. Bos. Side-Channel Marvels: Daredevil, 2016. URL: https://github.com/SideChannelMarvels/Daredevil.

[BB17] Paul Bottinelli and Joppe W. Bos. Computational aspects of correlation power analysis. *Journal of Cryptographic Engineering*, 7(3):167–181, September 1, 2017. URL: https://doi.org/10.1007/s13389-016-0122-9.

[BK16] Cees-Bart Breunesse and Ilya Kizhvatov. Jlsca, 2016. URL: https://github.com/Riscure/Jlsca.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, Lecture Notes in Computer Science, pages 16–29, Berlin, Heidelberg. Springer, 2004.

[BKM+20] Wolfgang Bubberman, Sengim Karayalcin, Matthias Meester, Olaf Braakman, and Stjepan Picek. Side-channel analysis toolbox, 2020.

[Cas22] Gaëtan Cassiers. SCABench: A suite of benchmarks for side-channel analysis libraries, 2022. URL: https://github.com/cassiersg/SCABench.

[CB23] Gaëtan Cassiers and Olivier Bronchain. SCALib: A side-channel analysis library. *Journal of Open Source Software*, 8(86):5196, 2023. URL: https://doi.org/10.21105/joss.05196.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, Lecture Notes in Computer Science, pages 13–28, Berlin, Heidelberg. Springer, 2003.

[CKN01] Jean-Sébasticn Coron, Paul Kocher, and David Naccache. Statistics and Secret Leakage. In Yair Frankel, editor, *Financial Cryptography*, Lecture Notes in Computer Science, pages 157–173, Berlin, Heidelberg. Springer, 2001.

[Dev23] Developers of h5py. Fancy indexing in h5py. October 9, 2023. URL: https://docs.h5py.org/en/3.10.0/high/dataset.html#fancy-indexing.

[DTO20] Alex Dewar, Jean-Pierre Thibault, and Colin O'Flynn. NAEAN0010: Power Analysis on FPGA Implementation of AES Using CW305 & ChipWhisperer, 2020. URL: https://media.newae.com/appnotes/NAE0010_Whitepaper_CW305_AES_SCA_Attack.pdf.

[Dre07] Ulrich Drepper. What Every Programmer Should Know About Memory, November 21, 2007. URL: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[eSh19] eShard. SCAred, 2019. URL: https://gitlab.com/eshard/scared.

[FHM+19] Alberto Fuentes Rodriguez, Luis Hernandez Encinas, Agustin Martin Munoz, and Bernardo Alarcos Alcazar. A Modular and Optimized Toolbox for Side-Channel Analysis. *IEEE Access*, 7:21889–21903, 2019. URL: https://ieeexplore.ieee.org/document/8636501/.

[GBT+08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual Information Analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, Lecture Notes in Computer Science, pages 426–442, Berlin, Heidelberg. Springer, 2008.

[GLP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. Stochastic Methods. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, Lecture Notes in Computer Science, pages 15–29, Berlin, Heidelberg. Springer, 2006.

[GJJ+11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for sidechannel resistance validation, 2011.

[HDF] HDFGroup. The HDF5® Library & File Format. URL: https://www.hdfgroup.org/solutions/hdf5/.

[HRG14] Annelie Heuser, Olivier Rioul, and Sylvain Guilley. Good Is Not Good Enough. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, Lecture Notes in Computer Science, pages 55–74, Berlin, Heidelberg. Springer, 2014.

[HMH+12] Johann Heyszl, Stefan Mangard, Benedikt Heinz, Frederic Stumpf, and Georg Sigl. Localized Electromagnetic Analysis of Cryptographic Implementations. In Orr Dunkelman, editor, *Topics in Cryptology – CT-RSA 2012*, Lecture Notes in Computer Science, pages 231–244, Berlin, Heidelberg. Springer, 2012.

[HMH+13] Johann Heyszl, Dominik Merli, Benedikt Heinz, Fabrizio De Santis, and Georg Sigl. Strengths and Limitations of High-Resolution Electromagnetic Field Measurements for Side-Channel Analysis. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science, pages 248–262, Berlin, Heidelberg. Springer, 2013.

[ISU18] Vincent Immler, Robert Specht, and Florian Unterstein. Your rails cannot hide from localized EM: how dual-rail logic fails on FPGAs—extended version. *Journal of Cryptographic Engineering*, 8(2):125–139, June 1, 2018. URL: https://link.springer.com/article/10.1007/s13389-018-0185-x.

[Int23] Intel. Intel VTune Profiler Performance Analysis Cookbook, 2023. URL: https://cdrdv2.intel.com/v1/dl/getContent/766317?fileName=vtune-profiler_cookbook_2023.0-766316-766317.pdf.

[KW13] Ilya Kizhvatov and Marc Witteman. Academic vs. industrial perspective on SCA. 2013. URL: https://cosade.telecom-paristech.fr/cosade13/presentations/session5a_f.pdf.

[Knu81] Donald Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Publishing Company, Reading, Mass, second edition edition, January 1, 1981. 624 pages.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, Lecture Notes in Computer Science, pages 388–397, Berlin, Heidelberg. Springer, 1999.

[LED18] LEDGER. LASCAR: Ledger's advanced side channel analysis repository, 2018. URL: https://github.com/Ledger-Donjon/lascar.

[LPR13] Victor Lomné, Emmanuel Prouff, and Thomas Roche. Behind the Scene of Side Channel Attacks. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, Lecture Notes in Computer Science, pages 506–525, Berlin, Heidelberg. Springer, 2013.

[LZ411] LZ4 developers. LZ4 - Extremely fast compression, 2011. URL: https://lz4.org/.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer US, 2007. URL: //www.springer.com/de/book/9780387308579.

[May00] Rita Mayer-Sommer. Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, Lecture Notes in Computer Science, pages 78–92, Berlin, Heidelberg. Springer, 2000.

[MM12] Amir Moradi and Oliver Mischke. How Far Should Theory Be from Practice? In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, Lecture Notes in Computer Science, pages 92–106, Berlin, Heidelberg. Springer, 2012.

[MM13] Amir Moradi and Oliver Mischke. On the Simplicity of Converting Leakages from Multivariate to Univariate. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, Lecture Notes in Computer Science, pages 1–20, Berlin, Heidelberg. Springer, 2013.

[MS16] Amir Moradi and Tobias Schneider. Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel Analysis and Secure Design*, Lecture Notes in Computer Science, pages 71–87, Cham. Springer International Publishing, 2016.

[Nat02] National Institute of Standards and Technology (NIST). *FIPS PUB 140-2: Security Requirements for Cryptographic Modules*. NIST, Gaithersburg, MD, USA, May 2002.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, Lecture Notes in Computer Science, pages 243–260, Cham. Springer International Publishing, 2014.

[OP11] David Oswald and Christof Paar. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, Lecture Notes in Computer Science, pages 207–222, Berlin, Heidelberg. Springer, 2011.

[ORP13] David Oswald, Bastian Richter, and Christof Paar. Side-Channel Attacks on the Yubikey 2 One-Time Password Generator. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, Lecture Notes in Computer Science, pages 204–222, Berlin, Heidelberg. Springer, 2013.

[PEK$^+$09] Christof Paar, Thomas Eisenbarth, Markus Kasper, Timo Kasper, and Amir Moradi. KeeLoq and Side-Channel Analysis-Evolution of an Attack. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 65–69, September 2009. URL: https://ieeexplore.ieee.org/abstract/document/5412857.

[PGA$^+$23] Kostas Papagiannopoulos, Ognjen Glamočanin, Melissa Azouaoui, Dorian Ros, Francesco Regazzoni, and Mirjana Stojilović. The Side-channel Metrics Cheat Sheet. *ACM Computing Surveys*, 55(10):216:1–216:38, February 2, 2023. URL: https://doi.org/10.1145/3565571.

[PSQ07] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Power and electromagnetic analysis: Improved model, consequences and comparisons. *Integration*. Embedded Cryptographic Hardware, 40(1):52–60, January 1, 2007. URL: https://www.sciencedirect.com/science/article/pii/S0167926005000647.

[PSA22] PSA Certified. The Turning Point for IoT Security – PSA Certified 2022 Security Report, 2022.

[QS01] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, Lecture Notes in Computer Science, pages 200–210, Berlin, Heidelberg. Springer, 2001.

[RGV17] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Fast Leakage Assessment. 2017. URL: https://eprint.iacr.org/2017/624. preprint.

[RWM19] Bastian Richter, Alexander Wild, and Amir Moradi. Automated Probe Repositioning for On-Die EM Measurements. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, November 2019. URL: https://ieeexplore.ieee.org/document/8942157.

[Ris18] Riscure. Inspector Trace Set .trs file support in Python, 2018. URL: https://github.com/Riscure/python-trsfile.

[RLM$^+$21] Thomas Roche, Victor Lomné, Camille Mutschler, and Laurent Imbert. A Side Journey To Titan. In pages 231–248, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/roche.

[Rot09] Arie Shoshani Rotem Doron, editor. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman and Hall/CRC, New York, December 14, 2009. 590 pages.

[Saa17] Saalfeld Lab. Not HDF5, 2017. URL: https://github.com/saalfeldlab/n5.

[SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - a clear roadmap for side-channel evaluations. 2015. URL: https://eprint.iacr.org/2015/207. preprint.

[SMG16] Tobias Schneider, Amir Moradi, and Tim Güneysu. Robust and One-Pass Parallel Computation of Correlation-Based Attacks at Arbitrary Order. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel Analysis and Secure Design*, Lecture Notes in Computer Science, pages 199–217, Cham. Springer International Publishing, 2016.

[SMS+17] Tobias Schneider, Amir Moradi, François-Xavier Standaert, and Tim Güneysu. Bridging the Gap: Advanced Tools for Side-Channel Leakage Estimation Beyond Gaussian Templates and Histograms. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, Lecture Notes in Computer Science, pages 58–78, Cham. Springer International Publishing, 2017.

[SW12] Sergei Skorobogatov and Christopher Woods. In the blink of an eye: There goes your AES key. 2012. URL: https://eprint.iacr.org/2012/296. preprint.

[SMK+17] Petr Socha, Vojtěch Miškovský, Hana Kubátová, and Martin Novotný. Optimization of Pearson correlation coefficient calculation for DPA and comparison of different approaches. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 184–189, April 2017. URL: https://ieeexplore.ieee.org/document/7934563?signout=success.

[SIU+18] Robert Specht, Vincent Immler, Florian Unterstein, Johann Heyszl, and Georg Sigl. Dividing the threshold: Multi-probe localized EM analysis on threshold implementations. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 33–40, April 2018.

[SMY09] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, Lecture Notes in Computer Science, pages 443–461, Berlin, Heidelberg. Springer, 2009.

[The09] The Blosc Development Team. Blosc, 2009. URL: https://www.blosc.org/.

[The06] The Common Criteria Recognition Agreement Members. Common Criteria for Information Technology Security Evaluation, September 2006. URL: http://www.commoncriteriaportal.org/.

[Tim19] Benjamin Timon. Scared: open source side-channel library by eShard. September 25, 2019. URL: https://eshard.com/posts/scared.

[UHD+17] Florian Unterstein, Johann Heyszl, Fabrizio De Santis, and Robert Specht. Dissecting Leakage Resilient PRFs with Multivariate Localized EM Attacks. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design*, Lecture Notes in Computer Science, pages 34–49, Cham. Springer International Publishing, 2017.

[VK12] Rajesh Velegalati and Jens-Peter Kaps. Introducing FOBOS: Flexible open-source board for side-channel analysis. In *Constructive Side-Channel Analysis and Secure Design (COSADE), Third International Workshop on: Work in Progress Session*. Citeseer, 2012.

[VS09] Nicolas Veyrat-Charvillon and François-Xavier Standaert. Mutual Information Analysis: How, When and Why? In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, Lecture Notes in Computer Science, pages 429–443, Berlin, Heidelberg. Springer, 2009.

[Yas14] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014. URL: https://ieeexplore.ieee.org/document/6844459.

[Zar15] Zarr Developers. Zarr, 2015. URL: https://zarr.dev/.