Whiteboxgrind – Automated Analysis of Whitebox Cryptography

Tobias Holl[1], Katharina Bogad[2], and Michael Gruber[3]

[1] Ruhr-Universität Bochum, Germany
tobias.holl@rub.de
[2] Fraunhofer Institute for Applied and Integrated Security, Garching, Germany
katharina.bogad@aisec.fraunhofer.de
[3] Technical University of Munich, Germany,
Chair of Security in Information Technology
m.gruber@tum.de,

Abstract. Digital intellectual property is often protected by encrypting the data up to the point of use. Whitebox cryptography is an attempt to provide users with the ability to decrypt that data without actually revealing the key by embedding the key inside a cryptographic implementation. In this work, we design and implement WHITEBOXGRIND, a fast, fully automated toolchain that obtains execution traces from whitebox implementations and applies DCA to recover the hidden embedded keys. To evaluate WHITEBOXGRIND, we analysed whiteboxes of the CHES WhibOx 2019 competition, and found WHITEBOXGRIND to provide a significant performance improvement over the state-of-the-art tooling, enabling attacks that were previously infeasible due to memory constraints. Furthermore, we provide WHITEBOXGRIND's source code.

Keywords: Whitebox, Differential Computation Analysis, Side Channel Analysis, CHES WhibOx

1 Introduction

When modern software needs to protect data from unauthorized access, cryptography is the only feasible solution. While encrypted communications generally allow for some form of key exchange or agreement, protecting data at rest requires storing an encryption key somewhere. This is easy if the software runs in a trusted environment (e.g. in a Trusted Platform Module (TPM) or on a corporate server), but not so straightforward in other cases (e.g. on devices controlled by end users).

So-called *whitebox* implementations combine a cryptographic algorithm with a fixed key, and add additional layers of obfuscation to hide the key from the user. Therefore, *whitebox* implementations violate Kerckhoffs's principle by design. Usually, these "whiteboxes" are used when the goal is to shield data or code from inspection by a potential adversary in an untrusted ecosystem (i.e. the user and their device respectively). This includes Digital Rights Management

(DRM), but is also found as a hardening mechanism against manipulation in other software that deals with confidential information (e.g. banking apps) [14].

The name *whitebox cryptography* already betrays that such a system is inherently insecure: Since all internal details are observable, it is always theoretically possible (though not always practically feasible) to reconstruct both the secret key and the cryptographic algorithm. Instead of preventing decryption entirely, whitebox cryptography can only serve as an obfuscation mechanism that discourages or delays a potential attacker by increasing the cost of an attack.

The straightforward approach to extract the key from a whitebox is direct reverse engineering. However, modern whiteboxes are sufficiently obfuscated to make this an extremely tedious and challenging task: Layers of obfuscation can be applied automatically, inflating the amount of code and data a reverse engineer needs to examine, while an attacker must manually understand and deobfuscate each layer.

Apart from glaring vulnerabilities in the cryptography itself, it is also essentially impossible to successfully attack such a system by merely observing and controlling its inputs and outputs, as modern cryptosystems are highly resistant to such attacks by design: For a cryptosystem to be considered even adequate, we generally require them to be resistant against both chosen-plaintext and (adaptive) chosen-ciphertext attacks [17], where attackers are able to make arbitrary queries to an *encryption* and *decryption oracle*—much like an attacker that only uses a whitebox implementation, but does not analyze its internals.

However, implementations of cryptosystems can still leak information due to a *side-channel*. By analyzing the time taken for a cryptographic operation (a *timing attack*), or the power consumed by the device during that time (e.g. via Differential Power Analysis (DPA) [19]), it can sometimes be possible to extract the key from a whitebox system. An active attacker that is able to manipulate the internal state of the cryptosystem while it is running has even more options (e.g. Fault Injection Analysis (FIA), where deviations in the output caused by a manipulation of internal state are analyzed).

In embedded systems, hardware protection mechanisms can require attackers who try to mount such an attack to make a significant upfront investment into specialized equipment. Software-only whiteboxes cannot rely on such hardening approaches: Since we can fully control the environment in which they are executed, it becomes much easier to isolate the whitebox implementation from the rest of the system. Sources of randomness which would ordinarily be used to hide internal values from an observer can easily be replaced with a determinstic stream of numbers. Here, hardware implementations can rely on a Cryptographically secure Random Number Generator (CSRNG) that is much harder to manipulate or replace. For software, we additionally have access to a large set of introspection tools such as debuggers and emulators that can be used to examine the internal workings of the whitebox in detail. This also means that side-channel attacks can be mounted not just on physical observations such as time and power consumption, but also on the internal behavior of the program during execution. The state-of-the-art equivalent to DPA for such whiteboxes is Differential Computation Analysis (DCA), which instead of deriving leakage information from the power consumption directly uses values extracted from traces of the program's execution [3, 4]. Because these types of attacks require large amounts of data obtained by continuously observating the whitebox implementation, they generally require some level of automation to be feasible.

Contribution In this work, we propose an automatic approach to efficiently collect and filter execution traces of whitebox implementations. In particular, we make the following contributions: We construct a fast tracer using Valgrind's [24] just-in-time-compilation abilities. We design a novel trace storage format that enables both fast processing and space-efficient storage. We implement our approach in WHITEBOXGRIND, a fully automated parallel DCA attack toolchain (trace collection, filtering, and the DCA), and benchmark it on several submissions to the 2019 *CHES WhibOx* contest [7]. While we target AES in this paper, the tracing and filtering stages of WHITEBOXGRIND generalize to arbitrary whitebox implementations; only the implementation of the attack itself needs to be adapted to the targeted cryptosystem. Also, we provide WHITEBOXGRIND's source code[4].

2 Related Work

There are several other tools that attempt to perform side-channel attacks on whitebox cryptography:

Frameworks Bos et al. [4], and Bock et al. [3] propose similar instrumentation-based approaches for tracing and applying Differential Computation Analysis (DCA), but without discussing any strategies to handle the vast amounts of data generated by their approaches. Both approaches are based on the idea of instrumenting the implementation with Valgrind [24] and recording the execution. Additionally, the same tooling can be used to perform FIA, though there is some difficulty in identifying the correct time and place at which a fault should be injected.

Sample reduction In order to reduce the amount of data that needs to be processed, Breunesse et al. introduced Conditional Sample Reduction (CSR) [5]. Our solution is less aggressive in discarding samples and cannot remove "superfluous" traces. We instead opt for an approach with much lower memory requirements that also requires less knowledge about the target implementation. We discuss the differences in more detail in Section 4.4.

Attack tools Finally, there are generic side-channel attack tools that process traces from arbitrary sources (if they can be brought into the right format). Examples include *LASCAR* [20] and *QSCAT* [13].

3 Background

We focus mainly on attacks against whitebox implementations of the Advanced Encryption Standard (AES). In the following, we briefly introduce the structure

[4] `https://gitlab.lrz.de/tueisec/whiteboxgrind`

of AES and the main concepts behind DCA, including its application to AES, and discuss various ways to obtain program traces.

3.1 Advanced Encryption Standard

The AES [12] is a 128-bit block cipher based on a substitution-permutation network (SPN). Like most block ciphers, it is composed of several near-identical rounds which consist from the following functions:
AddRoundKey adds the current round key $k^{(i)}$ to the state by a simple bitwise XOR ($\oplus$). This is equivalent to an element-wise addition in $GF(2^8)$.
SubBytes is a nonlinear byte-wise substitution where each byte a_i of the state is substituted via a constant look-up table to the output byte $b_i = S[a_i]$. The S-Box is carefully designed to make AES more resistant against various kinds of attacks [9], including differential and linear cryptanalysis [2, 22].
ShiftRows is a simple permutation of the bytes. When considering the state's 16 bytes as elements of a matrix in $GF(2^8)^{4\times4}$ (stored in column-major order), each row is rotated to the left one element further than the previous row, with the first row not being modified at all. This ensures that localized state changes propagate quickly to all state bytes (a desirable property for ciphers known as *diffusion*) [9, 10, 25].
MixColumns has a similar purpose by applying a linear transformation in $GF(2^8)$ to the individual *columns* of the same matrix.

3.2 Whitebox Cryptography

Cryptographic implementations where the key is configurable first derive the round keys $k^{(i)}$ from the cipher key k using the AES *key schedule*. In whitebox implementations, on the other hand, the key schedule is usually done in advance (since the key is already known at compile time) and is not present in the final implementation. Since the keys are fixed, the individual components of each round can be combined into a single table lookup (known as a T-box) [9]. If this lookup also includes the key addition, the values stored in the table can reveal the round key. To avoid this, *internal encodings* apply transformations to the input and output of such a table (or more generally of any internal operation). *External encodings* are similar, except that the transformations are applied to the plaintexts and ciphertexts outside the encryption algorithm [8, 21]. Usually, additional obfuscation is then applied to further hide the round keys from an observer.

3.3 Correlation Power Analysis

Differential Power Analysis (DPA) is a type of power analysis based on partitioning which was proposed in 1999 by Kocher et al. [19]. Brier et al. proposed a similar approach called Correlation Power Analysis (CPA) which uses correlation as a distinguisher in 2004 [6]. For a CPA attack, the power profile of a cryptographic operation is measured multiple times, e.g., in the context of AES different plaintexts with the same key. Subsequently, an intermediate value t, which depends on known values, and an unknown part (a single byte) of the key is chosen. A possible intermediate value t during the AES-*encryption* is the output of a first round's S-box. The intermediate value t is calculated as $t = S(k_0^{(0)} \oplus p_0)$ for all key hypotheses of the key byte $k_0^{(0)}$ The correlation of all key hypotheses with the measured power traces can be calculated after the application of a power model to the intermediate value. For the CPA of AES the intermediate value's Hamming Weight (HW) is commonly used as power model as it requires no prior knowledge about the state. The correct key byte is then determined by the hypothesis that yields the highest absolute correlation value. The principles of DPA were applied to whitebox cryptography by Bos et al. [4]. In the context of whitebox cryptography, DPA is referred to as DCA.

3.4 Program Tracing

Program traces are a useful tool in software analysis that allow us to draw conclusions about the underlying software's behavior (even if not much about the program is known, e.g. [11]). This means that there are quite a few different approaches by which we can obtain them:

Full emulation The idea behind fully emulating software is simple: we model hardware behavior as accurately as possible, and then simply run the software under analysis in the emulator. This allows us to observe all the internals of a program without having to understand it before. Constructing the emulator is a painstaking process, but only needs to be done once for a specific piece of hardware, and while the result is usually quite slow in terms of real-time performance, the data obtained is as close to the ground truth as one can get if the emulator is constructed accurately. Unfortunately, the performance penalty is usually quite severe, which is a problem for analyses like DCA that require *multiple* execution traces.

Hooking A much faster approach is to identify locations of interest in the program and modify the code at those locations to emit tracing events. However, accurately modifying the program without accidentally damaging functionality or missing out on some events can be difficult (especially if the implementation is one that hardened against reverse-engineering, like the ones we are dealing with in this work).

Debugger-based tracing Another common way to follow the execution of a program is to use a debugger's single-step feature: The debugger repeatedly signals the operating system to execute a single instruction at a time, after which control returns to the debugger, which can then inspect registers and memory. Here, we run into the opposite problem of the emulator: After an instruction has been executed, we need to understand *which* changes it made. Additionally, the frequent context switches between debugger and target come with a significant performance penalty.

Hardware-based tracing Some modern processors have features that allow constructing execution traces directly in the CPU. These features generally have low runtime overhead and access to ground-truth information, which are particularly desirable features for a program tracer. Unfortunately, they are often limited in scope (e.g. Intel's *processor trace* feature [15] only records control flow events, so the exact execution flow needs to be reconstructed after-the-fact, and information on memory accesses is missing entirely). Additionally, hardware-based tracing cannot usually intercept sources of nondeterminism (e.g. system calls), so elements from other tracing methods (debugging or hooking) will need to be borrowed—alongside their disadvantages.

Lifting and JIT A good compromise between the previous approaches is based on just-in-time compilation (JIT). The idea is to analyze a chunk of code that is about to be executed, *lift* it to an intermediate representation (IR) by carefully breaking down the CPU instructions into smaller operations, insert the code that logs the appropriate events, and then compile it back down to native code that is then executed. This lifting operation can be slow, but if pieces of code are executed multiple times, the results can be cached. Because the event logging is embedded directly into the native code, no expensive callbacks or context switches are necessary while the code is executing.

4 Whiteboxgrind

In the following, we describe our approach to efficiently trace and attack whitebox implementations, which we implemented in our Whiteboxgrind toolchain.

4.1 Trace Acquisition

We use a custom tool for the Valgrind framework [24] in order to obtain execution traces of a target whitebox when invoked with different inputs. It records every instruction executed during the cryptographic operation alongside all memory accesses.

We chose to base our tracer on Valgrind instead of one of the other approaches described in Section 3.4 for performance reasons: For every original instruction, at least one of our hooks is called to generate the instruction trace, plus hooks for every memory access. If the hooks are not implemented natively or not embedded directly into the execution stream, each hook invocation comes with a significant performance penalty.

Valgrind also uses a JIT-based approach, but allows us to manipulate the generated code on a lower level: When a basic block starts executing, Valgrind lifts it to its VEX IR by representing each instruction as a series of IR instructions, with particularly complex operations represented by calls to helper functions. Figure 1 shows how an example function is translated to VEX. On this intermediate representation, the active tool can perform arbitrary transformations. In Whiteboxgrind, we use this to insert our own instrumentation steps. Once control returns to Valgrind, the IR is compiled back down to native code and executed.

```c
uint8_t Sbox[256];
uint8_t Shift[16] = /* ... */;
void last_round(uint8_t state[16],
                uint8_t k[16]) {
  uint8_t t[16];
  for (int i = 0; i < 16; ++i)
    t[Shift[i]] = Sbox[state[i]];
  for (int i = 0; i < 16; ++i)
    state[i] = t[i] ^ k[i];
}
```

(a) Source Code

```
 1 mov    r9, &Shift
 2 mov    r8, &Sbox
 3 xor    eax, eax
 4 movzx  ecx, byte [&state+rax]
 5 movzx  edx, byte [rax+r9]
 6 inc    rax
 7 mov    cl, byte [r8+rcx]
 8 mov    byte [&t+rdx], cl
 9 cmp    rax, 0x10
10 jne    to line 4
```

(b) Assembly Code

```
---- IMark(Address of movzx, 5, 0) ----
t18 = GET:I64(r9)
t16 = Add64(t11,t18)
t22 = LDle:I8(t16)
    ≡ Ist_WrTmp(t22, Iex_Load(t16, 8bit, LE))
t48 = 8Uto32(t22)
t21 = t48
t49 = 32Uto64(t21)
t20 = t49
PUT(rdx) = t20
```

(c) VEX IR

Fig. 1: Example of the translation between source code, AMD64 assembly code, and the VEX IR

During instrumentation, we insert code to emit a program counter trace event on every `Ist_IMark` statement (which marks the start of a new instruction, hence the name), and a memory access event whenever the statement type or the type of a subexpression indicates that a memory access will take place. Within subexpressions, only memory reads can occur (`Iex_Load`). An example of this is the statement in Fig. 1c that is highlighted in green. We also log a memory read on `Ist_LoadG` statements if the associated guard condition is satisfied. Memory writes (`Ist_Store` or `Ist_StoreG` with a guard condition) and compare-and-swap instructions (`Ist_CAS`) are tracked separately. Beyond that, we need to handle calls to external helper functions that VEX inserts for more complicated instructions (`Ist_Dirty`) and load-linked/store-conditional (LLSC) statements (`Ist_LLSC`), both of which helpfully track their memory side effects in the VEX statement structure.

For all events, we store the value of the current instruction pointer or program counter. For memory accesses, we additionally store the target memory address, the value that is read or written, and the size or "width" of that value. Valgrind additionally provides us with the endianness of the access, though on most architectures this value is a constant. For compare-and-swap instructions, we store both the old value that is read for the comparison and the new value which is written if the comparison succeeds. Because we may later want to synchronize between the different types of events, each event is accompanied by an index that counts up as events are emitted, regardless of type.

In order to avoid tracing *all* of the target binary, we support isolating the encryption or decryption function. Depending on the scenario, this is done either using Valgrind's *client requests*, which allow the program under analysis to indicate to the tracer to start and stop tracing[5], or by starting and stopping tracing at user-provided addresses.

Between these points, the tracer ensures that execution of the traced binary is deterministic: System calls and other non-deterministic instructions such as `rdtsc` (returning the current processor timestamp) and `rdrand` (returning a random value) are reported to the user for manual patching, and can generally be replaced with "normal" instructions that return a constant value instead.

[5] We use this feature in combination with a custom harness for our evaluation in Section 5, where the encryption function is provided directly.

This means that the resulting traces can be compared across executions: The only reason why traces can differ from each other is that each run of the whitebox is provided with different inputs.

For convenience, WHITEBOXGRIND comes with a wrapper tool that manages the individual runs of the tracer. In particular, it takes care of input generation, configuring the tracer appropriately, collecting the trace events (via a Unix domain socket connected to the tracer), and finally storing the results.

4.2 Trace Storage

The traces generated in the previous step can be quite large (cf. Section 5). In order to perform further processing within reasonable limits on runtime and memory use, we need to store traces in a compressed format that still allows fast parallel access.

Unfortunately, existing formats either do not fully meet these requirements (e.g. the default implementation of HDF5 requires high-overhead locking to operate in a threadsafe manner, and the ParallelHDF5 variant only supports multi-*processing* rather than multithreading [26]), or are not designed with multidimensional data in mind (libraries such as *fst* [18] focus explicitly on two-dimensional data, while our traces—a matrix[6] of structured trace events with multiple attributes as described in Section 4.1—are essentially three-dimensional).

Therefore, we designed the y5 file format to address these shortcomings and enable efficient processing of our traces. Below, we briefly explain each of the design goals we considered during the development of the y5 file format, and how we achieved them.

Support for trace transposition Throughout our pipeline, we want to be able to process traces in parallel. Both during sample reduction (cf. Section 4.4) and in the actual attack (cf. Section 4.6), operations work on a set of matching samples, one from each trace. In a traditional storage format, this would mean storing the matrix of traces in column-major order (so that we can sequentially read columns of matching samples from the file). However, to create such a file, we would need to produce *all* traces at the same time, and ensure that all tracers generate events in a synchronized fashion, because it is impossible to know ahead of time how large each trace will be. This would significantly impact tracer performance and resource usage. Instead, our tracing framework stores one trace after the other (with traces as rows), and we transpose the matrix during the first sample reduction step in order to then be able to write processed columns of matching samples (turning traces into columns). Figure 2 shows how the dimensions of the data in a y5 file change during processing. The file format needs to accommodate this matrix transposition.

Compression Memory and control flow traces both contain highly redundant information. Values may be read multiple times, loops mean the same instructions occur repeatedly, and because Valgrind does not implement Address Space Layout Randomization (ASLR), memory addresses all share similar prefixes and

[6] Initially, each row of the matrix contains a full trace of the program.

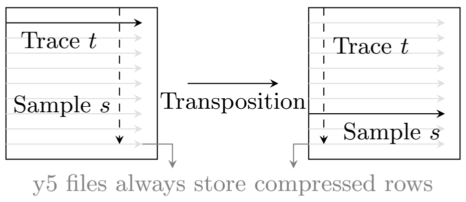


Fig. 2: Transposition of traces during processing.

remain the same across traces. To reduce the size of our traces while stored on disk, we need to compress the data in a way that ensures fast compression and decompression as well as significant size reduction. We use Google's *Brotli* compression algorithm [1] at a medium compression level to achieve a trade-off between these two goals. Each row, regardless of whether that represents a full trace or an aligned set of samples across all traces, is compressed separately to allow independent decompression without having to first process additional rows.

Fast seeking While it is useful to be able to decompress each row separately, we still need to locate the start of the row in the file. To do this, we prefix each row with its compressed size. This means we can easily skip each row during processing. However, this still makes seeking to the nth row a slow operation on larger trace files with many rows. In addition to the row headers, we maintain a fixed-size Table of Contents (TOC) at the start of the file[7]. Initially (while the number of rows r is less than the number of entries t in the table), each row has its own TOC entry. As more rows are added, entry t starts representing row $2r$, then $4r$ and so on. This allows us to fairly swiftly skip a large part of the file before following the row headers to finally locate a target row.

Low memory footprint Most software relies on a common file processing paradigm: read the compressed data into a buffer, decompress it into another buffer, and then process the data in that buffer. Because the operating system already caches the compressed file contents in memory, this pairing of reading and decompressing essentially stores the file contents in memory twice. We instead rely on a streaming approach: We directly map a segment[8] of the compressed file into memory. Then, as data is requested for processing, we extend that segment at the end (to be able to decompress more of the file), while shrinking it from the front to remove already-decompressed parts from memory. Because of this, we only keep a fixed amount of compressed data in RAM, and the user can manage how much of the decompressed data they want to request at any given time.

Support for parallel access Writing to a file in parallel without knowing the size in advance is essentially impossible, because the offsets at which each thread should write are not known ahead of time. On the other hand, *reading* files in parallel is only limited by the fact that the storage device usually does not sup-

[7] In our implementation, the size is configurable, but only at the time the file is created.

[8] The size of this segment is configurable to make parallel processing less memory-intensive, while optimizing single-threaded performance by reducing the number of mapping requests that need to be made.

port parallel access. Here, the fact that we also need to decompress the data means that even though the device may need to serialize our access requests, the subsequent decompression of different rows can be performed in parallel, which significantly improves read speeds. Using memory-mapped IO is very helpful here, because this allows us to maintain multiple windows into the file at different offsets, while normal file descriptor-based APIs expect there to be a single canonical offset into a file at which reading is performed.

Figure 3 shows the layout of a y5 file. Fields that are fixed at file creation time are hatched, the others can change as more data is added to the file. Each row represents eight bytes of data; multi-byte fields are packed in little-endian format for faster processing on x86 CPUs (i.e. most commonly available hardware). We provide a C++ library for UNIX-like systems[9] as well as low-level Python bindings using *pybind11* [16] in order to interface with existing software.

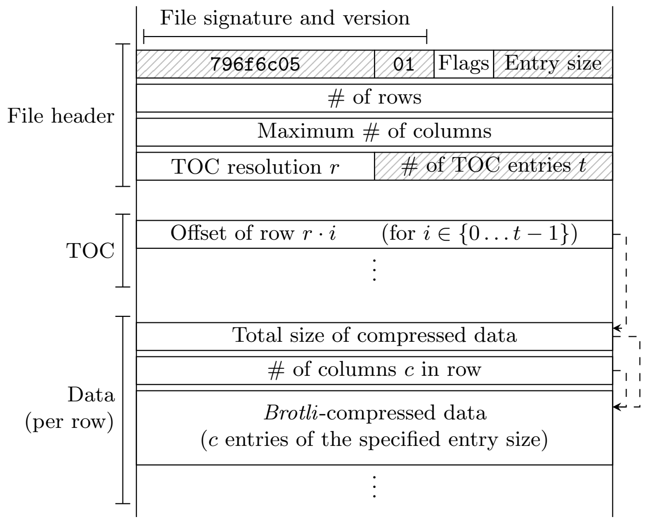


Fig. 3: Structure of a y5 file.

4.3 Parallel Architecture

Each of our tools consumes a y5 file. Depending on the input format and use case, we read either chunks of rows or chunks of columns in parallel by distributing the individual compressed rows into a thread pool. The task scheduler in charge of reading collects the decompressed data and passes it along to a processing pipeline.

[9] We are not aware of any constraints that would make a native Windows implementation impossible, but do not currently support Windows' memory-mapped IO functions.

Each pipeline step takes place in a separate thread, optionally distributing parallel implementations of "slow" tasks (including any processing step that needs to iterate over the individual trace events in the data) to a thread pool that can scale to an arbitrary number of CPUs. By transferring data ownership directly, we can avoid expensive locking operations.

Finally, any output chunks are written back to a y5 file. To ensure proper ordering, each chunk is accompanied by its index. In the (rare) case that chunks do arrive out of order, later chunks are held back until the missing chunk arrives. To limit total memory consumption caused by uneven processing speeds (e.g. if reading is faster than the processing, the chunks would "pile up" while waiting for that stage of the pipeline to clear up), we restrict the total number of chunks in processing at any given point in time.

4.4 Sample Reduction

Obfuscation measures in whitebox implementations can inflate the total number of instructions and memory accesses significantly, especially if—as in [7]—it is specifically designed to resist analysis. This intentionally added complexity helps defend against manual reverse-engineering efforts and seriously harms the ability of automated tooling to process the entire implementation, both in terms of code analysis (e.g. for decompilation) and with regards to the traces that we use. In essence, we have *too much* data to efficiently perform the attacks described in Section 3.3. To reduce the size of the traces that we analyze, we can rely on two observations:

Non-data-dependent events First, trace events that remain the same (including the value that is read or written in a memory access) along all recorded traces usually do not depend on the input data[10]. We can discard these events.

Repeated events Second, repeated occurrences of the same set of trace events across all traces (e.g. repeated memory reads from the same address without the value being modified inbetween) can be deduplicated, since the attacks do not take structures across multiple trace events into account. This is equivalent to the *duplicate column removal* described in [5].

If the traces are properly aligned (i.e. there is no data-dependent execution that causes the same part of the cryptographic algorithm to yield a variable number of trace events depending on the input), both of these cases can be filtered out[11]. For non-aligned traces (e.g. where the length of the trace depends on the input), more sophisticated filtering approaches are needed. We do not currently handle non-aligned traces.

Whiteboxgrind uses separate tools to remove events that match either of the two criteria described above.

[10] Assuming a random distribution of inputs, the probability of this not being the case is generally low in terms of the number of traces.

[11] This is not always the case. However, data-dependent execution that depends on intermediate values directly leaks information about those values, which can then be used for similar attacks. Whiteboxgrind does not currently implement this.

To reduce the number of passes over the raw data, we first remove non-data-dependent events during the initial transposition step described in Section 4.2. Unlike Conditional Sample Reduction (CSR) [5], we do not attempt to remove "superfluous" traces that will not yield additional information during the attack stage: CSR partitions the trace matrix into groups depending on which input values can result in the value found in a specific observation. Then, samples that observe inconsistent (i.e. different) values for the same partial input can be discarded. Similarly, traces with inputs in the same partition can be deduplicated. However, this requires the user to select a partitioning function/bit mask to choose relevant input bits, and has a $\mathcal{O}(n)$ memory requirement (linear in the trace size), which can become prohibitive for the lengths involved in our evaluation (see Section 5). Instead, we filter out samples where the value does not depend on *any* of the input bits. Problems with random masking can be avoided by substituting random sources with constant values (by intercepting system calls and relevant machine code instructions such as `rdrand` in the tracer).

Perfectly identifying repeated events for deduplication in theory requires us to keep the entire set of known events in memory (which is infeasible given the size of some of our traces) or to repeatedly search the events we have already emitted (at an unacceptable $\mathcal{O}(n^2)$ complexity). Instead, we use a hash-table-based least-recently used (LRU) cache that can grow up to a fixed size, letting us efficiently identify duplicates that are "close enough" to each other in time[12]. Because the attacks implemented by WHITEBOXGRIND act on individual samples, non-removed duplicates increase the execution time of the subsequent steps, but do not impact the final result [5].

4.5 Visualization

Understanding the internal structure of a whitebox implementation from a program trace by hand is difficult without some form of visualization. Plotting the program counter over time in a scatter plot reveals how the code is structured, and doing the same for memory accesses reveals the layout of the data (both of the intermediate values and of constants such as S- or T-boxes).

During our research, we observed that the traces generated by WHITEBOXGRIND can be too large to comfortably load into RAM and visualize even after sample reduction[13]. To remedy the situation, we implemented a *renderer on steroids* that uses the DirectX Direct2D API [23] to draw the trace diagram.

Further than using a GPU to render the plot, we also introduced algorithmic optimizations. Traditional rendering programs usually scale the data in a lossless way to the screen, compressing neighbouring points together until all data fits into the available space. This results in a high-quality, high-accuraccy plot of

[12] A randomized cache eviction policy would allow us to remove further events without increasing the cache size, but the added reduction in event count we observed during our evaluation using this policy (even repeatedly) was marginal.

[13] Existing tools generally insist on doing this; we can only speculate as to why this is the case.

the data, that retains the full shape even if scaled down. For our use case, we are only interested in the macro shape of the collected traces—consequently, we can skip the expensive compression step under the assumption that trace entries are indicative of their neighbours. Intutively, this is at least the case for the program counter plot: While our data is in no form steady, we argue that within basic execution blocks some properties of steadiness are retained (mainly, that if no jump instruction is encountered the next instruction is neighbouring the currently executed instruction).

Then, we can restrict loading to a subset of the whole data and assume that the general shape of the plot remains unchanged. Implementation-wise, we realize this by pre-computing the available space of the plot in pixels, and distributing these pixels evenly across the time axis of the plot. We then use a windowing approach to pan, zoom and scale the displayed data.

A drawback of this method is its inherent loss of some data. Unlike algorithms based on compression, our plotter may miss spikes in execution, giving a false impression of the overall shape if these spikes are small enough.

However, we argue that this is not a problem for our use-case. Our down-sampling approach suppresses noise spikes if they are sufficiently small, but in general preserves the overall shape of the plot. Barring sudden jumps, we intuitively compare our algorithm to audio recording with varying sample rates, where sample rate reductions are audible, but even very low sample rates retain enough information that a human can recognize a recording.

Furthermore, in our testing we found that while the plot-invisible noise might be the parts we are interested in to recover the key, the plot is more meaningful when applied to filtered data. With the accompanying reduction in overall plot size, less down-sampling needs to be done for low zoom levels, which improves the plots accuracy. Concerning the larger structures we are interested in (e.g. to identify the different rounds in an AES implementation). We can see in Fig. 4 the internal structure of the `distracted_leavitt` whitebox from [7]: there, six separate loops (easily discernible from the image) of nine iterations each (note the slightly larger distance between these iterations) process the current state in four 32-bit blocks in a T-box-esque implementation.

4.6 Attack

Once the traces are pruned to a manageable size, WHITEBOXGRIND applies a standard DCA approach to recover the key. We apply a leakage model (by default, we use the HW, though this is easily replaced if desired) to the values read from or written to memory by the whitebox and use the results as our side-channel leakages.

Using a user-configurable selection function on either the plain- or ciphertext of the values recorded alongside the traces (cf. Section 3.3), we compute the hypothesis values $\mathbf{H} \in \mathbb{R}^{256\times t}$ (one value for each possible key byte and each of the t program traces), and center the values around the origin $\mathbf{h}_k = \mathbf{H}_k - \overline{\mathbf{H}_k}$ for each of the key byte values $k = 0 \dots 255$. This processing only needs to be performed once for each attack run and can be done ahead of time.

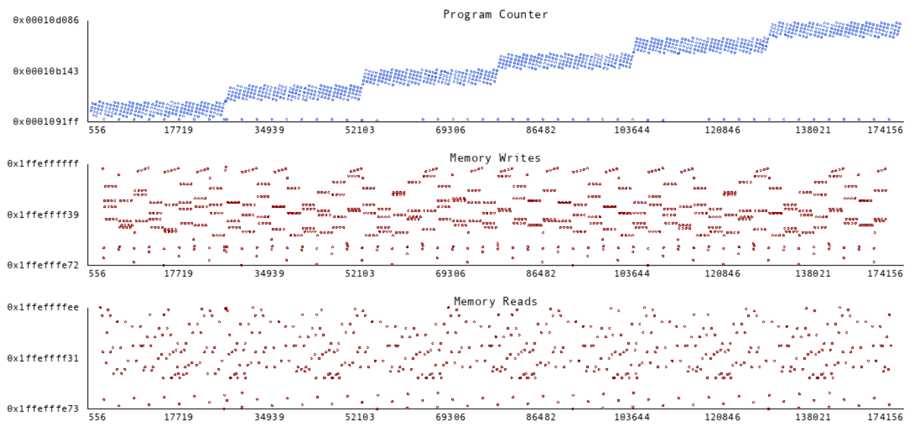


Fig. 4: Access patterns of the `distracted_leavitt` whitebox from [7]

As chunks of traces arrive (cf. Section 4.3), we normalize each set of events in parallel. Given our leakage matrix $\mathbf{T} \in \mathbb{R}^{s \times t}$ consisting of s samples across t separate program traces[14], we compute the centered values $\mathbf{t}_i = \mathbf{T}_i - \overline{\mathbf{T}_i}$ for each sample, and the Euclidean norm $\|\mathbf{t}_i\|$.

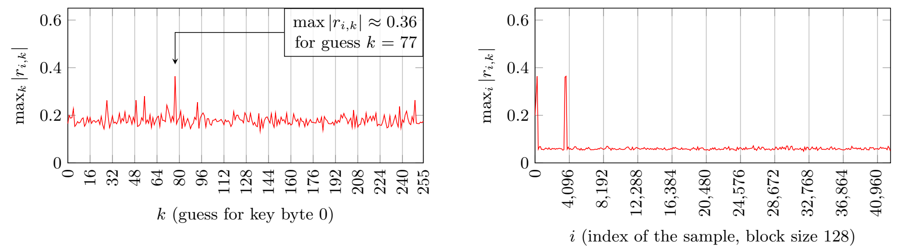
Now, computing the Pearson correlation coefficient between the hypotheses obtained from the selection function and the leakage values from each of the trace events is simple:

$$r_{i,k} = \frac{\mathbf{t}_i \cdot \mathbf{h}_k}{\|\mathbf{t}_i\| \|\mathbf{h}_k\|}$$

It is sufficient to store $\operatorname{argmax}_k |r_{i,k}| \, \forall i \in \{0 \ldots s\}$ in order to obtain the final key (of course, to compute the argmax value, we also need to store the corresponding maximal $|r_{i,k}|$). Because we want to allow for *some* visualization of the correlation we instead store $\max_i |r_{i,k}| \, \forall i \in \{0 \ldots s\}, k \in \{0 \ldots 256\}$ (at processing-block-level resolution). If the attack is successful, the correct key byte value should show a peak when plotting the 256 resulting correlation values. Figure 5a shows such a case. Similarly, plotting the maximum correlation against the index of the sample where it is obtained (Fig. 5b) shows where that key byte is processed. In this case, we are targeting the first round.

Note that this deviates from the behavior of tools such as *LASCAR* [20], which attempt to store the full matrix of *all* $r_{i,k}$ (which is, of course, suboptimal given the amount of data that needs to be processed). The impact of this optimization is examined further in Section 5.

[14] In practice, elements of $\mathbf{T}$ are of course not from $\mathbb{R}$; rather, common Hamming weight leakages are in $\mathbb{Z}/256\mathbb{Z}$, i.e. a single byte value—but after normalization, they are treated as floating-point values. Mathematically, we assume they are in $\mathbb{R}$, and simply accept some small level of error in the practical computations.



(a) Peak correlation $\max_k |r_{i,k}|$ for each k (across guesses)

(b) Peak correlation $\max_i |r_{i,k}|$ for each i (across samples)

Fig. 5: Correlation for key byte 0 of `peaceful_williams` [7]

5 Evaluation

We evaluated WHITEBOXGRIND on a set of 7 whiteboxes with no data-dependent execution from the 2019 *CHES WhibOx* contest [7] and a "textbook" reference AES implementation with a hard-coded key that was not hardened against side-channel attacks. For each implementation, we collected execution traces for the same set of plaintexts that was randomly selected prior to the evaluation. For this evaluation, we used the values that are read from memory (as opposed to values written to memory or those involved in compare-and-swap operations, for which we obtain separate traces) as our side-channel.

Table 1 shows the time taken by each of WHITEBOXGRIND's individual tools on the traces generated by the first 100 inputs to each of the targets alongside the number of samples per trace in the resulting outputs. Note that this is generally insufficient to perform a successful attack, but serves nicely to illustrate the performance differences between approaches. We should note that while it is possible to obtain traces in parallel and concatenate the y5 files afterwards, we did not do this for this evaluation. All performance measurements were taken on a machine with an AMD EPYC 7552 CPU with 96 threads and 1TB of total RAM (at 3200 MT/s and CL22). To avoid high IO latencies on physical disk accesses, we stored all data on a 300GB RAM disk.

The time taken for any operation generally scales with the size of the traces, but the correlation is not fully linear. Besides the amount of data involved, performance can also depend on other implementation characteristics (e.g. for the tracer, some instructions are far less efficient after VEX translation than others).

The initial 100 traces collected for our evaluation were sufficient to recover the correct key from the unprotected reference implementation, but not from any of the other implementations. For those implementations where collecting traces was reasonably fast, we collected additional traces to prove the correctness of our implementation. At 500 traces, we were also able to recover the key from the (obfuscated) `peaceful_williams` whitebox.

Additionally, we compared WHITEBOXGRIND's runtime performance (again at 100 traces) to that of the CPA implemented in *LASCAR* [20]. Figure 6 shows the performance improvements we achieve over *LASCAR*'s implementation.

Implementation	**Tracing**	**Sample reduction**		**Leakage**	**Attack**
		Non-data-dependent	Repeated		
Reference	00:00:16	00:00:00	00:00:00	00:00:00	00:00:00
	1191	992	992		
`distracted_leavitt`	00:00:36	00:00:02	00:00:02	00:00:01	00:00:03
	26020	14113	13951		
`elegant_turing`	06:54:14	01:14:21	00:42:49	00:17:09	01:04:58
	72928481	27971488	17215702		
`flamboyant_engelbart`	01:19:22	00:16:49	00:06:29	00:02:52	00:10:46
	17232291	5733448	2981379		
`friendly_edison`	13:13:53	05:01:08	04:02:02	01:30:41	05:45:18
	214383342	141535907	91556105		
`goofy_archimedes`	00:00:18	00:00:00	00:00:00	00:00:00	00:00:00
	4412	2413	2386		
`goofy_lichterman`	05:56:55	00:55:13	00:35:22	00:13:27	00:52:04
	53746357	21207273	12968282		
`peaceful_williams`	00:01:13	00:00:07	00:00:05	00:00:02	00:00:10
	241761	57645	42525		

Table 1: WHITEBOXGRIND performance (time and number of samples per trace) on AES whiteboxes from [7]

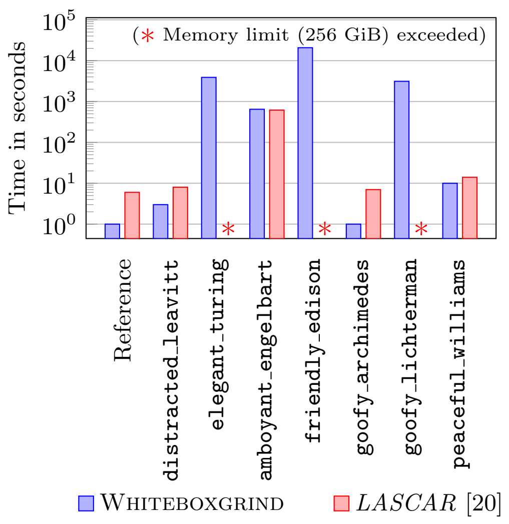
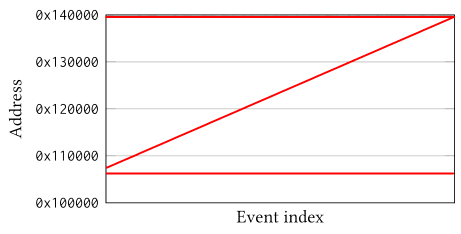


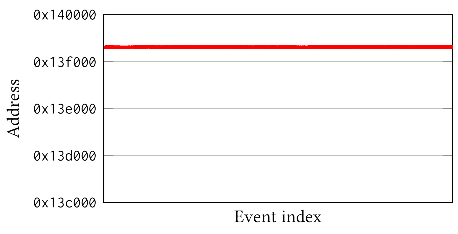
Fig. 6: Performance of WHITEBOXGRIND's CPA compared to *LASCAR* [20]

Finally, we analyzed the impact of our sample reduction strategies (cf. Section 4.4) and of the data compression in the y5 file format (cf. Section 4.2):

On the WhibOx implementations [7], we observed a reduction in size of between 29.27% (for `friendly_edison`) and 76.16% (for `peaceful_williams`) by removing samples that do not differ between inputs, and up to 48.00% (for `flamboyant_engelbart`) by removing recurring samples. As an example, Fig. 7 shows how the two sample reduction steps affect the traces obtained from the `peaceful_williams` whitebox. `peaceful_williams` implements AES by means of a virtual machine that processes a hardcoded instruction stream. This is the steadily rising line in Fig. 7a. Together with the line at the bottom (accesses to a compiler-generated jump table), these accesses do not depend on the input, and are filtered out.



(a) Memory reads before filtering



(b) Memory reads after filtering

Fig. 7: Sample reduction for `peaceful_williams` [7]

Because the trace data is highly structured, y5's compression was able to reduce the total file size of all eight memory read traces from Table 1 before sample reduction by 91.88% (from 2137 GiB to 174 GiB). On the transposed traces after filtering, the compression ratio is similar (91.92%, from 1232 GiB to 99 GiB). Only after the leakage is computed (where only a single byte is stored per entry) does the average compression ratio drop below 90% (however, at that point the amount of data is already significantly lower than at the start). Figure 8 shows the compression ratios after each processing step.

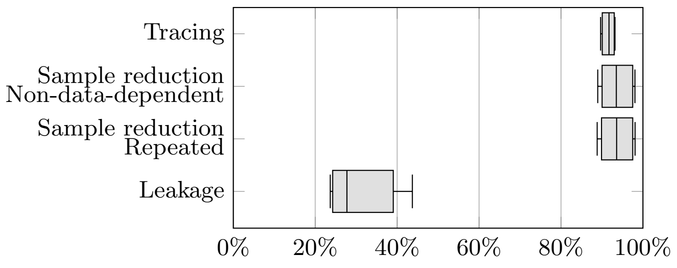


Fig. 8: Compression ratios using y5 on traced memory reads from Table 1.

6 Discussion

We believe that Whiteboxgrind proves that a fully automated analysis of many whitebox implementations is feasible given sufficient computing resources. Unless an implementation is specifically hardened to avoid side-channel attacks based on memory values (e.g. by relying on register values only, which can easily be countered by adjusting our instrumentation strategy), it will usually be possible to use the intermediate values observed to draw conclusions about the key that is used. During our evaluation, we found that we were mostly constrained by the time taken to obtain and process the large amounts of data included in our traces. An attacker not constrained by research budgets will generally be able to efficiently obtain more traces by applying more computational power — then, the main performance constraint becomes the processing speed during sample reduction. In practice, an attacker will also be able to select "interesting" parts of the traces depending on the selection function (e.g. if targeting the first round, one might only want to consider the first half of the trace) instead of processing the entire trace, further speeding up subsequent processing steps. Generally, the large variance between the different implementations stems from the different obfuscation strategies chosen by these submissions.

One possible way to defeat the approach described in this work is to introduce data-dependent execution that causes a misalignment between different traces (if possible, without leaking intermediate values). This can be achieved by explicit dependencies on the original plaintext, which does not need to be kept secret. A more sophisticated approach to re-aligning the traces—perhaps based on algorithms normally used for line-based text diffing, or those used for other side-channel attacks (e.g. [27])—might enable successful attacks on those obfuscation schemes.

7 Conclusion

In this work we examined whether it is possible to fully automate DCA against whitebox implementations of AES. Of course, this involves making some trade-offs with regard to performance and attack results. Notwithstanding these particularities, we achieved an attack speed increase of roughly a magnitude on commodity hardware against a reference AES implementation. We stress that this performance measurement needs to be taken with a grain of salt: for about half of our whitebox samples, existing tooling yielded no result at all due to practical space constraints. This work should be considered as yet another sign that whitebox cryptography is a fundamentally broken approach: Given that it is possible to fully automate key extraction for some of the less hardened approaches, most implementations will not hold up to a dedicated attacker with reverse engineering skills. We suspect that it will be possible to apply sufficient obfuscation to make a fully automated approach infeasible to the point where the tools need to be adjusted to the specific target by hand. However, real-world implementations usually have to conform to more requirements than such theoretical designs, which means that the automated analysis approach presented in this work will still hold value for many such implementations. This also applies to non-AES whiteboxes which are vulnerable to DCA-style attacks. Our method of obtaining and filtering instruction traces does not assume any specifics about the implementation under test, so that only minor modifications to the attack code are required for WHITEBOXGRIND to support attacks on other cryptosystems.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions on the paper, as these helped us to improve it. This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the SIPSENSIN project under grant number 16KIS1663 and by the German Research Foundation (DFG) under the Excellence Strategy - EXC 2092 CASA - 390781972. This work was done while Tobias Holl and Katharina Bogad were at the Technical University of Munich.

References

1. Jyrki Alakuijala and Zoltan Szabadka. Brotli Compressed Data Format. RFC 7932, July 2016.
2. Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. *Journal of Cryptology*, 4(1):3–72, January 1991.
3. Estuardo Alpirez Bock, Joppe W. Bos, Chris Brzuska, Charles Hubain, Wil Michiels, Cristofaro Mune, Eloi Sanfelix Gonzalez, Philippe Teuwen, and Alexander Treff. White-Box Cryptography: Don't Forget About Grey-Box Attacks. *Journal of Cryptology*, 32(4):1095–1143, February 2019.
4. Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In *Lecture Notes in Computer Science*, pages 215–236. Springer Berlin Heidelberg, 2016.
5. Cees-Bart Breunesse, Ilya Kizhvatov, Ruben Muijrers, and Albert Spruyt. Towards fully automated analysis of whiteboxes: Perfect dimensionality reduction for perfect leakage. Cryptology ePrint Archive, Report 2018/095, 2018. `https://ia.cr/2018/095`.
6. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg, 2004.
7. CHES 2019. WhibOx contest. Online, August 2019.
8. Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, pages 250–270. Springer Berlin Heidelberg, 2003.
9. Joan Daemen and Vincent Rijmen. The Rijndael Block Cipher. AES Proposal, March 1999.
10. Joan Daemen and Vincent Rijmen. The Wide Trail Design Strategy. In *Cryptography and Coding*, pages 222–238. Springer Berlin Heidelberg, 2001.
11. Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. ACM Press, November 2013.

12. Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Federal Information Processing Standards Publication 197: Advanced Encryption Standard (AES), 2001-11-26 2001.
13. “FdLSifu”. QSCAT — Qt Side Channel Analysis Tool. Online [retrieved 2022-04-28], 2017–2021.
14. Vincent Haupert, Dominik Maier, Nicolas Schneider, Julian Kirsch, and Tilo Müller. Honey, I Shrunk Your App Security: The State of Android App Hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 69–91. Springer International Publishing, 2018.
15. Intel Corporation. *Intel® Architecture Instruction Set Extensions Programming Reference.* Intel, 2021.
16. Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – Seamless operability between C++11 and Python, 2017. https://github.com/pybind/pybind11.
17. Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* Chapman & Hall/CRC, 1st edition, 2008.
18. Marcus Klik et al. The fst format and fstlib library. Online [retrieved 2022-04-21], April 2019.
19. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology — CRYPTO' 99*, pages 388–397. Springer Berlin Heidelberg, 1999.
20. Ledger Donjon. Lascar: Donjon Side Channel Library. Online [retrieved 2022-04-21], February 2019–2022.
21. Tancrède Lepoint and Matthieu Rivain. Another nail in the coffin of white-box aes implementations. Cryptology ePrint Archive, Report 2013/455, 2013. `https://ia.cr/2013/455`.
22. Mitsuru Matsui and Atsuhiro Yamagishi. A New Method for Known Plaintext Attack of FEAL Cipher. In *Advances in Cryptology — EUROCRYPT' 92*, pages 81–91. Springer Berlin Heidelberg, 1993.
23. Microsoft. Direct2d api. `https://docs.microsoft.com/en-us/windows/win32/direct2d/direct2d-portal`.
24. Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007.
25. Claude Shannon. A Mathematical Theory of Cryptography. Memorandum, Bell Laboratories, September 1945.
26. The HDF Group. HDF5 Application Developer's Guide. Online [retrieved 2021-10-07], September 2019.
27. Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA – differential address trace analysis: Finding address-based Side-Channels in binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 603–620, Baltimore, MD, August 2018. USENIX Association.