Towards Fully Automated Analysis of Whiteboxes

Perfect Dimensionality Reduction for Perfect Leakage

Cees-Bart Breunesse[1], Ilya Kizhvatov[2*], Ruben Muijrers[1] and Albert Spruyt[1]

[1] Riscure, {breunesse,muijrers,spruyt}@riscure.com
[2] Digital Security Group, Radboud University Nijmegen, i.kizhvatov@cs.ru.nl

Abstract. Differential computation analysis (DCA) is a technique recently introduced by Bos et al. and Sanfelix et al. for key recovery from whitebox implementations of symmetric ciphers. It consists in applying the differential power analysis approach to software execution traces that are obtained by tracing the memory accesses of a whitebox application. While being very effective, DCA relies on analyst intuition to be efficient. In particular, memory range selection is needed to prevent software execution traces from becoming prohibitively long. Moreover, analyst failure to specify the relevant range lets the vulnerable whitebox implementation be evaluated as secure.

We present a novel approach for dimensionality reduction of software execution traces, that takes a significant part of analyst intuition out of the loop. The approach exploits the lack of measurement noise in the traces and selects only the samples that are relevant for the key recovery. Our experiments with the published whitebox implementations show that the length of software execution traces can be automatically reduced to a few dozens of bits. This results in an attack speedup of several orders of magnitude, which in turn facilitates the use of more computationally intensive DCA flavours such as multiple leakage targets proposed by Klemsa.

Our approach simplifies the methodology for whitebox analysis down to the tracing of a large default memory range, letting our dimensionality reduction techniques extract the relevant points for DCA, and run the attack on multiple leakage targets, excluding analyst errors and saving analysis time. It also provides quick insights in case of whitebox implementations with additional protection layers such as encodings, and can be used to identify the range for fault injection in differential fault analysis. We make our techniques available to the community as a part of a free/libre open-source side channel analysis toolkit. We believe they are a step forward for fully automated whitebox analysis tools.

Keywords: whitebox · cryptanalysis · security evaluation · tools

1 Introduction

Whitebox implementations [CEJVO02] are a solution for secure key storage in software. They have gained adoption in practice specifically in the mobile payment and digital content protection applications, being a cost-efficient alternative in the situation where secure key storage in hardware is not available or comes at a high cost. The lack of practical security of pure whitebox solutions is currently compensated by the use of additional layers of protection, as well as by rolling out regular updates [JR02].

Very effective attacks on whiteboxes have been recently demonstrated by Bos et al. [BHMT16] and Sanfelix, Mune and de Haas [SMdH]. Essentially, they bring differential

*The research of the author was funded by the Netherlands Organisation for Scientific Research (NWO) through the projects 13499 TyPhoon and 628.001.005 OpenSesame.

power analysis (DPA) techniques long known in the hardware domain, to the whitebox setting, under the term Differential computation analysis (DCA) coined in [BHMT16]. DCA deals with software execution traces that are acquired by registering memory accesses of a whitebox implementation over the time of its execution. Many public whitebox implementations have succumbed to DCA and have been gathered in the Deadpool collection [Sid].

Our motivation. In this work we approach whitebox security specifically from a practical evaluator's perspective (which is very close to the attacker perspective). In the existing analysis methodology proposed in [BHMT16] and implemented in [Sid], an analyst applying DCA needs to manually configure the range of memory to trace in the acquisition phase, and to choose the right attack target also known as leakage target inside the implementation in the computational phase of the attack. These choices are crucial because software execution traces can be prohibitively long for the computational part of the attack that consists of the computation of Pearson's correlation coefficient.

Here an issue arises: the methodology requires a priori assumptions (i.e. intuition) about the implementation. If the assumptions are wrong, the key can not be (fully) recovered, this is the case for at least two implementations in the Deadpool collection of whiteboxes [Sid].

We would like to eliminate the need for the human input in whitebox analysis, and make analysis run fast. An obvious starting point is to look for an efficient approach to reduce the number of samples per software execution trace.

Our contribution. We present a novel approach for dimensionality reduction of software execution traces that eliminates a significant part of the analyst intuition in whitebox analysis and saves analysis time. The approach consists of two complementary techniques and an additional visualization method. Our techniques exploit the fact that software execution traces exhibit perfect leakage (i.e. are free from measurement noise) and therefore lend themselves to dimensionality reduction (point-of-interest selection) that would be impossible in the noisy DPA setting.

Our first technique, Duplicate Column Removal (DCR), is an efficient algorithm for the elimination of repeated columns and their inverses in the measurement matrix. The algorithm takes as input rows and not columns and has a smaller worst-case complexity than the well-known hash table based approaches, which makes it more suitable in the DCA setting than existing solutions[1]. The technique is generic in the sense that it does not require any knowledge of the cryptographic algorithm or the implementation.

Our second technique, Conditional Sample Reduction (CSR), is a novel proposal to eliminate columns of the measurement matrix that do not exhibit key-related dependencies. Namely, for a varying part of the input used to attack a part of the key, it eliminates the columns where samples do not vary. This requires the specification of the attack target, but as this information is needed for DCA, no additional knowledge is required. Counterintuitively, with CSR the attack becomes more efficient with an increase in traces (to a certain extent). The outcome of CSR provides a way to quickly identify the presence of additional countermeasures such as encodings or misalignment.

For a given software execution traceset, the combination of DCR and CSR leaves only the points that are relevant for key recovery, and no redundant points. This is why we call it perfect dimensionality reduction.

We also suggest a novel differential technique to visualize the propagation of input data throughout the execution of the whitebox. The visualization allows effective filters to be constructed as it gives insight into the operation of whiteboxes. In addition to DCA, it can be used to identify the range for fault injection in differential fault analysis.

As a result, we drastically reduce the number of samples in an automated manner.

[1] E.g. solutions based on existing functions in scientific computing languages: `duplicate()` in R or `unique()` in MATLAB or numpy.

For example, for a typical whitebox implementation we reduce the amount of samples by a factor of 8000 (from about 800×10^3 to about 100). This yields a several orders of magnitude speedup of the computational part of the attack. Such speedup in turn facilitates the execution of the attack on multiple leakage targets without the need to choose the right ones based on intuition or detailed knowledge of the implementation. For instance, Klemsa [Kle16] has described 255 different flavors of a leakage target, which with our approach can be attacked in the same time as a single target in the previous methodology. Our case study shows that this enables us to fully break an implementation that was not fully broken before.

Impact. Our techniques significantly reduce the amount of human input in the process of whitebox analysis. As a result, the methodology for whitebox analysis is simplified to the tracing of a large default memory range, letting our dimensionality reduction techniques extract the relevant points, and running DCA on multiple leakage targets. This lowers the chance that a vulnerable whitebox implementation is evaluated as secure. In addition, it shows that whitebox implementation without additional protection (or after additional protection was bypassed) can be broken significantly faster, and is therefore extremely hard to counteract with regular updates. We believe that our techniques are a step forward for fully automated whitebox analysis tools.

Related work. The most related technique is implemented by Teuwen in an attack on the MoVfuscator challenge [Teu15]. There, he identifies constant columns based on 10 traces, and then removes these columns from the entire measurement matrix. This allows to reduce the number of samples from 4×10^6 to 6.6×10^3.

The previously published approach of [BHMT16], implemented in [Sid], relies on manual memory range selection with some degree of automation based on known memory sections. The Daredevil tool of [Sid] deals with the significant length of software execution traces by memory-efficient and parallel implementation of Pearson's correlation as described in [BB17].

Our CSR technique is inspired by the conditional leakage averaging introduced for the DPA setting by Lomné, Prouff and Roche in [LPR13]; that technique provides reduction for the number of traces but does not select points of interest and therefore is inefficient in the DCA setting.

Implementation. To demonstrate the efficiency of our techniques, we implement them, apply them on a number of public whitebox implementations from the Deadpool collection, and compare them to the state-of-the-art attacks implemented in that collection in two dedicated case studies. The implementation is available as a part of a free/libre open source side channel analysis toolbox [Jls]. Datasets and scripts for reproducing our results are available in [Exp].

2 Background

In this section we provide a brief background to white box implementations from an attacker's perspective, with the focus on Differential Computation Analysis and its specifics compared to the classical DPA. For a more in-depth background on symmetric whitebox cryptography and attacks see e.g. [CEJVO02, BHMT16, SMdH, Kle16].

2.1 White Box Cryptography

Without hardware backed security, (mobile) digital rights management and payment applications need to use other mechanisms against adversaries who have fully compromised the system where these applications run. In these cases countermeasures need to be present in the mobile application itself to prevent credential theft.

In this model, called the whitebox-attacker model [CEJVO02], an attacker has access to all the data-at-rest and all the intermediate calculations. To protect against such attackers, whitebox cryptography was introduced [CEJVO02]. In whiteboxes the key is merged into the implementation of the cryptographic algorithm. This merging is typically performed by transforming the algorithm into successive table lookups into which the key is embedded. Typically, whiteboxed versions of T-DES and AES symmetric algorithms are developed; whiteboxed versions of asymmetric algorithms also exist, but to our best knowledge are not publicly available.

All public whitebox implementations have known attacks (see [BHMT16] for an overview). However, whitebox cryptography can still add a layer of protection in practice, if it is coupled with other mechanisms. These mechanisms include: obfuscation to discourage any manual reverse engineering; anti-debugging to deter dynamic analysis; anti-rooting to make analysis on live targets more difficult; anti-emulation to prevent running in an environment under complete attacker control; anti-lifting to prevent the whitebox from being taken and used as-is.

The first attacks on whiteboxes targeted whitebox schemes, but not implementations, for instance, the BGE attack [BGE04]. Though attacks of that kind had practical time complexity, they required relatively detailed knowledge of the scheme. A serious change in the whitebox security scene took place with the adoption of implementation attacks long known in the hardware world. In particular Differential Fault Analysis (DFA) [SMdH, JBF02], and Differential Power Analysis (DPA) [BHMT16, SMdH]. The latter, in its application to the whitebox setting, was termed Differential Computation Analysis. The advantage of these techniques is that they require significantly less knowledge of the whitebox scheme.

2.2 Differential Computation Analysis

DCA [BHMT16] is essentially very similar to DPA. The main difference is that observations of power consumption are replaced by observations of memory accesses. These observations are acquired by tracing the running whitebox using application instrumentation, and are called software execution traces. A software execution trace is a series of bits representing memory accesses in successive CPU cycles.

In DPA the number of acquired power traces is relatively large in order to deal with the measurement noise. In DCA, the number of acquired software traces is typically low (from about twenty to about a thousand), as there is no physical noise. But the number of samples per trace is specifically large if one considers the memory space of a whitebox application. As in DPA, most of these samples are not relevant for the key recovery as they are not related to the processing of key-dependent data.

To speed up the subsequent computational part of the attack, one has to reduce the number of samples per software execution trace during or after the tracing. The existing DCA methodology of [BHMT16], implemented in [Sid], applies filtering for a specific address region, or selects least significant bits from recorded 32- or 64-bit words. The filter ranges are typically chosen based on which segment they are from (e.g. heap, stack, bss or text). Alternatively, software execution traces are plotted and regions which show structures relating to the targeted algorithm are used to define monitored regions [BHMT16]. Though default filters exist, this methodology generally requires understanding of the implementation, and in particular analyst intuition from prior experience with whitebox implementations.

In the computational part, DCA applies a distinguisher to compare the observations of a certain target variable t dependent on the chunk of the key k and plaintext (or ciphertext) chunk x, with predictions of that target variable for all the values of the key chunk. The target variable is computed using a target function f: $t = f(x, k)$. For example, in the attack targeting the output of the first round AES S-box, $f(x, k) = \text{SubBytes}(x \oplus k)$. With

enough observations, the highest value of the distinguisher reveals the correct key chunk. Pearson's correlation coefficient [BCO04] or difference-of-means test [KJJ99] are used as a distinguisher.

As DCA deals with perfect leakage, physical leakage modeling for predictions of the target variable is not required. But the target variable still needs to be chosen. The target variables used in [BHMT16, Sid] are: the output of a first round S-Box for AES and DES; the Rijndael multiplicative inverse in the finite field of 2^8 elements (inside the first round S-Box) for AES. In [Kle16], Klemsa generalized the multiplicative inverse approach to considering all the possible multiplicative inverses for the AES S-box.

A variant of the attack is to apply a distinguisher to several target variables (e.g. every bit of an S-box output) and then use the sum of absolute values of individual distinguishers to determine the correct key chunk candidate. This approach is referred to as Absolute-Sum DPA (AS-DPA) [DPRS11]. Apart from AS-DPA with bit-wise targets, in this work we will use AS-DPA with the Klemsa targets, combining results for all the multiplicative inverses. For certain implementations this approach is more effective at the cost of additional computational effort.

2.3 Encodings and Desynchronization

Attacks on whiteboxes can be inhibited by countermeasures, in particular encodings and desynchronization. We briefly describe these countermeasures in order to describe their effect on our proposed techniques later.

An encoding is the representation of data in the whitebox scheme. Two kinds of encodings can be distinguished: intermediate and external. Intermediate (internal) encodings are so called mixing bijections [CEJVO02] applied to intermediate values inside the whitebox to obfuscate the lookup tables. If these encodings are however too linear, the whitebox will still be vulnerable to DCA [Kle16]. External encodings are bijective encodings applied on the input and output of algorithm. As a result, the implementation no longer conforms to standard cryptographic algorithms and is less interoperable, which inhibits the adoption of these encodings. DPA/DCA and DFA are not applicable standalone to properly implemented external encodings, as the attacker does not know the actual input/output data to compute the intermediate values.

It is essential that the traces are synchronized, because correlation or difference-of-means distinguishers work sample-wise. Namely, the samples corresponding to the processing of the DCA target should be in the same position across different traces. Countermeasures borrowed from the classical DPA field, such as randomized shuffling of operations, inhibit differential attacks by breaking synchronization across different acquisitions [BBIJ17].

2.4 Tools and Implementations

The Side-Channel Marvels project [Sid] contains a collection of whitebox implementations, termed Deadpool, as well as a toolbox for whitebox analysis.

The Deadpool collection contains a number of public whitebox implementations which we use as targets in this work. Contrary to what the name suggests, not all of these whiteboxes are fully broken by the accompanying DCA tools. Specifically, the `aes_ches2016` and `aes_plaidctf2013` whiteboxes are not fully broken. Later in this work we explain why Daredevil and Tracer did not do so, while the methodology using our techniques can efficiently recover the keys from these whiteboxes.

Very recently, a number of whiteboxes were provided in the CHES 2017 Capture The Flag contest [whi17]. All of them were broken during the contest.

The Side-Channel Marvels toolbox presents a coverage of tools required for whitebox analysis. It contains tools for acquiring execution traces based on Pin [pin] and

Valgrind [val]. It also contains a tool for visualizing traces that can aid in the reverse-engineering of whiteboxes. Once traces have been acquired they can be attacked using the Daredevil tool that efficiently implements the correlation-based distinguisher.

Another open-source tool for DCA is Jlsca [Jls], a high-performance DPA toolbox written in Julia. Similarly to Daredevil it is a tool to run the computation part of DPA and DCA attacks. Dimensionality reduction techniques presented in this work are integrated in Jlsca.

2.5 Notation

In the following sections different algorithms will be described. This is done in textual form as well as in pseudo code. In both forms the following notation is used:

- Scalars are denoted with a lower case letter e.g. x
- Sets are denoted with an upper case letter e.g. S and the subscript which element of the set is being used e.g. S_x
- Logical set operations are used for trivial operations e.g. assigning 0 to all elements: $\forall x \in \mathbb{N}, S_x \leftarrow 0$
- Algorithmic for loops are used for more complex set operations, however they are considered identical to the aforementioned notation.

In order to keep the descriptions concise, all set operations are assumed to operate from lowest index to highest index. All algorithms work on the set of traces O which is a bit matrix of size $n \times m$ where n is the number of traces and m is the number of samples (bits) per trace. Each entry $O_{i,j}$ is a single bit representing sample j from trace i. We also refer to O as the measurement matrix.

3 Duplicate Column Removal

In this section we describe duplicate column removal, a lossless compression technique for DCA measurement matrices that runs in linear time in trace length m.

3.1 Description

The rationale behind DCR is as follows: DPA-like attacks are vertical attacks, meaning that they work independently for each column in O. This means that if two columns contain the same values, the result of the attack (e. g. the value of Pearson's correlation coefficient) will be the same. Interesting to note is that when two columns x and y are each other's inverse ($O_{x,j} = 1 - O_{y,j}$ for all j), Pearson's correlation coefficient is inverted too. As in our analysis setting, it is the absolute value of the correlation coefficient which determines the attack result. We can remove all columns that are duplicates or inverse duplicates without any loss of information.

Our approach to duplicate removal is to process matrix O row by row and keep track of *groups* of duplicate columns. We start with the most general assumption: all columns are duplicates of the first column. Every time a new row is processed, the assumptions so far can be shown to be incorrect, e. g. two columns can turn out not to be duplicates. In such case a refinement step is performed: the violating column is removed from its group and placed in a new one, or added to an existing one if that set already exists. Note that two rows that are not duplicates can never become duplicates by processing more rows. After the last row is processed, all columns have been assigned a group and columns in

each group are identical. In a second pass over the data, all but one column is removed from each group.

For removing inverse duplicate columns, we introduce an additional data structure that keeps track of which columns are inverses of each other. Keeping track of inverses happens in a similar way to keeping track of duplicates. A significant difference is that in this case the initial assumption is not made before processing the first trace, but right after. After processing the first trace, there will be 2 groups (value 0 and value 1) which are each other's inverse. The same principle holds as for duplicate columns: if two columns are not each other's inverse at some point they can never become each other's inverse. This is why the refinement strategy is applicable. Algorithm 1 formally describes the full DCR technique.

Algorithm 1 Duplicate Column Removal

Require: n the number of traces
Require: m the number of samples
Require: $O_{i,j} \in 0, 1$ where $i < n$ and $j < m$ the value of sample j in trace i
Require: G_j where $j < m$ the group column of j
Require: N_j where $j < m$ the new group column of j if sample j had a different value than the group column in the current row
Require: I_j where $j < m$ the group column of the inverse group of j if j has an inverse

1: $\forall j < m, G_j \leftarrow 0$ ▷ initial assumption, all are duplicate of 0
2: $I \leftarrow \emptyset$
3: **for all** $i < n$ **do**
4: $N \leftarrow \emptyset$
5: **for all** $j < m$ **do**
6: **if** $O_{i,j} \neq O_{i,G_j}$ **then** ▷ violation detected
7: **if** N_{G_j} not set **then**
8: $N_{G_j} \leftarrow j$ ▷ new group with j as group column
9: **if** $i = 0$ **then**
10: $I_j \leftarrow 0$ ▷ First trace; define the initial inverse columns
11: **end if**
12: **end if**
13: $G_j \leftarrow N_{G_j}$ ▷ set group column for j
14: **end if**
15: **end for**
16: **for all** $\{j, k \mid I_j = k\}$ **do**
17: **if** $O_{i,j} = O_{i,k}$ **then** ▷ columns are no longer inverses
18: $I \leftarrow I - I_j$
19: **if** N_j is set **then**
20: $I_{max(k,N_j)} \leftarrow min(k, N_j)$
21: **end if**
22: **if** N_k is set **then**
23: $I_{max(j,N_k)} \leftarrow min(j, N_k)$
24: **end if**
25: **else if** N_j is set **and** N_k is set **then**
26: $I_{max(N_j,N_k)} \leftarrow min(N_j, N_k)$ ▷ old and new columns are inverses
27: **end if**
28: **end for**
29: **end for**
30: **return** $\{x \mid G_x = x$ **and** I_x not set $\}$

3.2 Properties

Due to the fact that only redundant information is removed, the results of the DCA attack will be the same with or without sample reduction. However, when applying DCR it is important to note the following:

1. Patterns or macro structures, which can for example reveal where the cipher rounds are in the data, often appear due to the repetition of values, e. g. memory addresses. Due to the nature of this algorithm, these patterns disappear after sample reduction. It is therefore recommended to make use of these patterns, for region selection or alignment, before applying DCR.

2. A similar warning holds for concatenation of tracesets (vertical stacking of two or more measurement matrices) from which DCR has been independently applied: DCR is not guaranteed to remove the same samples from the individual tracesets.

3. Although compression rates are typically quite high (see section 6), there are theoretical limits. Since only duplicates and inverses are removed, it is not possible to remove all constant samples (such samples will be eliminated by CSR). If there exists at least one constant sample in the original data set, exactly one will exist in the reduced set. Additionally, the difference in the result of removing only duplicates versus removal of duplicates together with their inverses is at most 50%. The removal of duplicates ensures that every column is unique after compression. So even if every column has an inverse, at most 50% of the columns can be removed by inverses removal.

4. If we increase the number of traces, DCR will normally remove less columns.

3.3 Comparison to Existing Alternatives

In Table 1 we compare DCR to the three well-known alternative implementations for duplicate column removal: a naive comparison and two hash table algorithms. Important to note is the assumptions made in how the data is processed. In DCA (and DPA), during acquisistion the measurements are naturally saved one by one, and measurement matrix O is therefore stored in the row-major layout. If the data needs to be presented in a column-major layout, a rotation of the matrix must be performed to avoid excessive cache thrashing. Matrix rotation is however, not a cache friendly operation. The naive comparison compares every column against every other column in O. The hash table algorithms store columns in a hash table in an attempt to quickly detect collisions. However, in order to not throw away columns from O that contain leakage, the columns that are stored in the hash table must be stored in full to allow comparison. This can be problematic when n is large.

Table 1: Runtime and memory comparison of various algorithms for removal of duplicate columns

Algorithm	Direction	Memory	Runtime (avg, worst)	Notes
Naive	column	$\mathbf{O}(nm)$	$\mathbf{O}(nm^2)$, $\mathbf{O}(nm^2)$	Requires O in memory, and rotation
Hash table	column	$\mathbf{O}(nm)$	$\mathbf{O}(nm)$, $\mathbf{O}(nm^2)$	Requires rotation, and collision resolution
Hash table	row	$\mathbf{O}(nm)$	$\mathbf{O}(nm)$, $\mathbf{O}(nm^2)$	Requires O in memory, and collision resolution
DCR	row	$\mathbf{O}(m)$	$\mathbf{O}(nm)$, $\mathbf{O}(nm)$	Free from collisions

We also see that the hash table implementations have a quadratic worst case running time. This worst case running time occurs when many columns hash to the same hash bucket

but are not identical. If two columns hash to the same hash bucket, the implementation needs to compare the samples of both columns to determine whether they are actually the same or merely hash collisions. One or more full column comparisons are required for each time a hash collision happens. In addition to the extra work, this behavior results in sub-optimal cache usage.

When the assumption is dropped that O will fit in memory, which is often the case for real-life whitebox implementations, only the per-column hash table based algorithm and DCR as presented here will be able to run without swapping parts of O to/from disk.

4 Conditional Sample Reduction

For DPA on noisy measurements of power usage or EM emanations, Lomné et al. [LPR13] have introduced conditional leakage averaging to reduce the number of observations, i.e. rows of O, so that the DPA distinguisher function runs on less traces and, therefore, faster than in the straightforward case. Inspired by that approach, we propose a novel technique for the case of perfect leakage that reduces the number of both rows and columns of O.

4.1 Description

Our approach is, in terms of Section 2.4 in [LPR13], a partitioning of O according to the possible input values of the target function used in the attack. Our key idea behind the partitioning is that for a given target function $f(x, k) = t$ with x being a part of the input, under a fixed chunk of the key k that we are recovering in the attack, the target variable t only changes when x changes. Since whitebox observations O are noiseless bits, we can discard columns in O where the bit value is different between two rows of O that have the same input, since in this case the target function f outputs the same intermediate.

This strategy allows for the reduction of the number of measurements for the same reason as conditional leakage averaging. But more importantly, in the perfect leakage setting, it allows for a reduction in the number of samples (bits), or the width of the measurement matrix, which is where this approach differs from conditional leakage averaging.

The formal description of CSR is given in Algorithm 2. The algorithm iterates over all rows of O and the corresponding inputs X to the target function f, building a set C of indices to unique input values, and a boolean mask M indicating which columns of O contribute to the leakage. Then, M and C are used to produce the output, consisting of a compressed measurement matrix O' and corresponding unique inputs X'. CSR complexity is $\mathbf{O}(nm)$ time and $\mathbf{O}(m)$ memory, where n is the number of traces and m is trace length (in bits). A step-by-step example of CSR on a test vector is given in Appendix A.

The algorithm needs to be run individually for every target function in the attack, except for the cases where the input to a group of target functions is the same (consider for instance individual bits of an S-box output). To attack AES with longer than 128-bit keys or DES and TDES inner rounds one needs to rerun the CSR algorithm using the inputs of the inner round target function.

For the combination with DCR, the order of the algorithms is in general not important. In our experiments, we first apply DCR on the original measurement matrix, then apply CSR to the DCR output. The output of CSR is fed into a distinguisher such as Pearson's correlation coefficient or difference-of-means to perform the key recovery.

4.2 Properties

As more traces (with random inputs) give more chance for similar input values to arise, CSR favors more traces. As our case studies in section 7 illustrate, the increase in the

Algorithm 2 Conditional Sample Reduction

Require: n the number of traces
Require: m the number bits per trace
Require: $O_{i,j} \in 0, 1$ where $i < n$ and $j < m$, observation matrix
Require: $X_i \in 0, 1$ where $i < n$, target function input vector, each element contains the target input for the corresponding row in O
Require: S set of possible target input values

Part 1: initialize helper arrays

$C, \forall b \in S, C_b \leftarrow \emptyset$. ▷ C_b will contain an index in X where input b is first seen.
$M, \forall j < m, M_j \leftarrow 1$. ▷ Boolean mask for the columns in O to keep

Part 2: compute the CSR mask

for all $i < n$ **do**
  if $C_{X_i} = \emptyset$ **then**
    $C_{X_i} \leftarrow i$
  end if
  $c \leftarrow C_{X_i}$
  for all $j < m$ **do**
    $M_j \leftarrow M_j \wedge \neg(O_{i,j} \oplus O_{c,j})$
  end for
end for

Part 3: apply the CSR mask

$n' \leftarrow$ number of non-empty elements in C
$m' \leftarrow$ number of non-zero elements in M
$X' \leftarrow X_C$ ▷ Create an array of unique target input values
$O' \leftarrow O_{C,M}$ ▷ Create an array of corresponding compressed traces
return O', $'X'$, n', m'

number of traces required for CSR to work effectively is compensated by a decrease of several orders of magnitude in the number of samples remaining after CSR.

There are situations where CSR may remove all the columns of the recorded bits matrix O, thus preventing any further analysis. This occurs in the following cases.

1. The measurements are misaligned, either due to the binary tracer not correctly filtering data dependent memory accesses, or due to active countermeasure in the whitebox implementation.

2. The target whitebox uses masking. Note that for masking, one needs secure random number generator, which is difficult to achieve for whiteboxes, since the environment in which the whitebox runs, is under complete attacker control.

3. The target whitebox uses encodings that are injections. These encodings, from n to m bits where $m > n$, may use more than one value to represent one non-encoded, plain, value. Columns in the recorded matrix that represent individual bits of such encodings will be eliminated by this approach, since they may vary between recordings with the same input.

Even in cases where an encoding may mitigate DCA by removing any linearity between the target intermediate and encoding, we note that the conditional sample reduction still works, except that it reduces the observation matrix to only contain the columns storing the encoded intermediates. For an example see Eloi Vanderbeken's `aes_nsc2013` in Deadpool. A DCA attack will fail on these reduced observations, but it points an attacker to the location (i.e. CPU instruction pointer) of this data. This aids in reverse engineering and could be used in combination with a DFA targeted to the identified location.

5 Differential Plots

Previously described plots such as those in [DGLHL13] and [Sid], can aid in the analysis of whiteboxes. We show an improvement on this method which allows simple identification of regions which are: not input dependent, input dependent lookup tables, or intermediate state.

Using memory instrumentation, we collect multiple traces from a whitebox. In [Sid], this could be based on Intel Pin or Valgrind. Depending on the required scenario the input is chosen in different ways: either random inputs or chosen plain texts. If the analyst only needs to identify the round state, random inputs are used. If the analyst wishes to gain additional insights into the whitebox, inputs are chosen in the following manner: a reference input is selected at random for the first run and successive whitebox executions are performed with single bit differences.

By using the second method, an analyst can identify how the input changes are diffused across the state. To illustrate the method, we perform differential plotting on the `aes_openwhitebox_chow` [CEJVO02]. We take 17 traces of this whitebox: a reference trace and 16 traces which have one byte of the input changed. This allows us to produce differential scatter plots, showing differences in single bytes of the input.

Figure 1 shows four differential scatter plots. The plots show a relatively small area in both time (horizontal axis) and address space (vertical axis). The time range corresponds to the execution of approximately 3 AES rounds. The address range covers slightly more than the 16 bytes of the AES round state. Memory accesses are labeled by markers. Marker type distinguishes reads and writes. Marker color indicates whether the address and the data are the same or different in each trace. Green markers have the same data and the same addresses, and can therefore be discarded. Purple markers indicate input dependency.

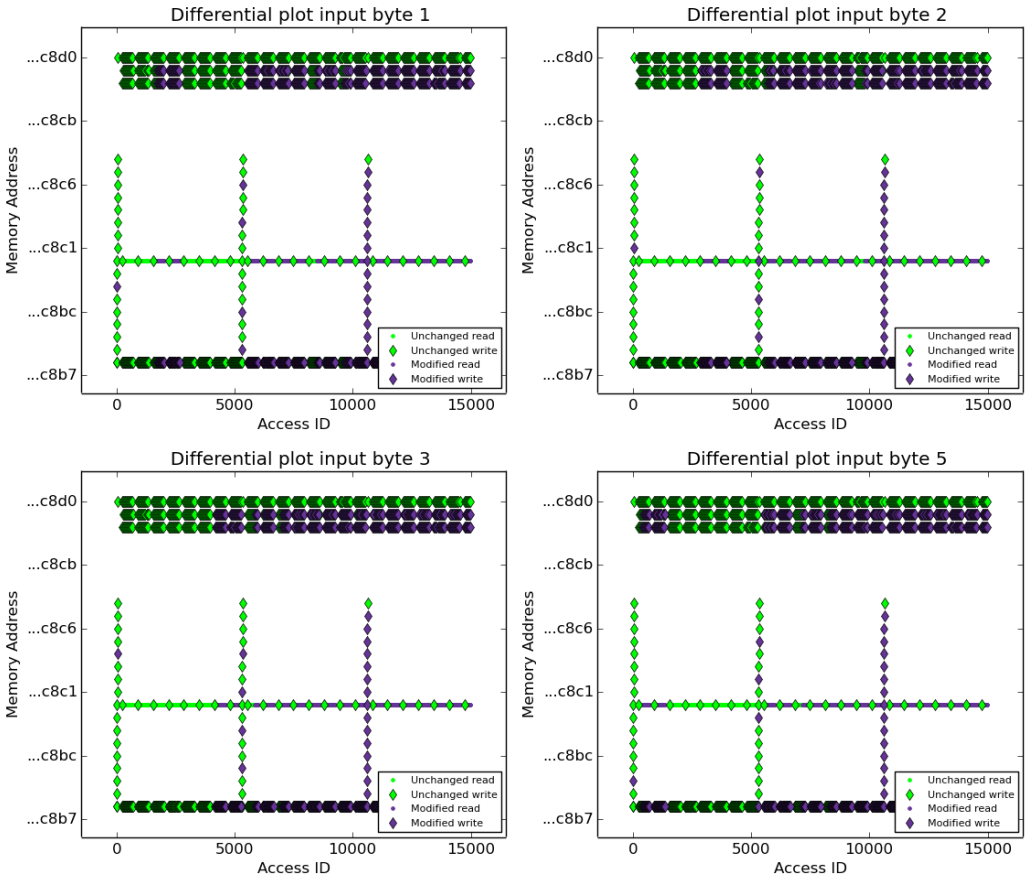


Figure 1: Differential plots for `aes_openwhitebox_chow`

Each of these plots shows changes which correspond to the expected changes after an AES MixColumn operation. In the first round, the difference appears only in one byte. In the second round, that difference diffuses to 4 bytes, due to the MixColumn operation. In round three, all bytes of the state have been affected. An analyst can use these plots to retrieve only the round state (e.g. the first column of accesses in the plots) and perform DCA on it. Note that even though only the targeted range is shown in Figure 1, this range can be found by visual inspection of the memory accesses over the entire whitebox execution time.

Similar plots can be used to narrow down the search space for DFA. Faults can be introduced into either the round state or the round keys. By making differential plots between the original trace and the fault trace, further structures can be identified, for instance DFA countermeasures. The algorithm as it is presented will fail to give the desired results if any misalignment is still present.

6 Experimental Results

Here we show the experimental results for DCR and CSR performed on the Deadpool collection [Sid] (in its state of June 2017). We only look at the number of samples; the full attack metrics will follow in section 7 in two dedicated case studies. Our results can be reproduced with the datasets and scripts available in [Exp].

Table 2 shows the original traces length in bits, and how many bits are left after

processing 200 traces. For DCR, the column *No dup* shows how many samples are left after duplicate removal, and column *No inv* shows how many are left after duplicate removal and inverse duplicate removal. The *CSR* column represents the samples left after CSR and DCR have been performed. The number of samples after CSR is given in bits left per chunk of the attacked key.

Table 2: Number of samples (bits) left after successive reductions

White box	Trace length	No dup	No inv	CSR
`aes_ches2016` (P)	804,248	49,913	49,894	97–105
`aes_hacklu2009`	8,896	1,506	1,505	11
`aes_karroumi2010` (P)	436,744	38,340	34,083	69–110
`aes_kryptologik`	33,544	18,431	18,427	33
`aes_nsc2013_variants` (P)	775,840	19,396	18,718	41
`aes_openwhitebox_chow`	131,176	16,258	16 257	41
`aes_plaidctf2013` (P)	37,440	7,037	6,989	19
`des_sstic2012`	1,824	1,058	1,057	15
`des_wyseur2007`, S-box out (P)	2,548,744	24,637	20,421	18–43
`des_wyseur2007`, round out (P)	2,548,744	24,637	20,421	36–152

While for instance [BHMT16] show that a whitebox implementations can often be broken in less than 50 traces, this is not normally how an analyst performs an attack. In order to gain confidence that the correct key has been recovered, more traces are taken to ensure that the correlation score for the correct key is significantly higher than the next key candidate (a metric formally referred to as the relative distinguishing margin, see [WO11]). Once the correct key is recovered with sufficient confidence, the amount of traces required to bring all the correct key candidates to the top is determined. For this reason, we do not include the acquisition time in our analysis as the number of needed traces is invariant.

For some whiteboxes, a modified Pin plugin was used, which did not filter the results in any way. These are marked (P). For `aes_ches2016` and `aes_plaidctf2013`, the Daredevil tool used in Deadpool did not fully recover the key on the original traces. Using the traces from our modified Pin plugin and the Klemsa leakage model, it becomes possible to recover the key, as we capture relevant samples and our reduction techniques do not remove them from the traces.

The implementation in `aes_kryptologik` is not a normal AES implementation, it is similar to AES256, as it contains 14 rounds but uses a custom AES key schedule. We provide numbers for recovering a single round key.

DES implementation in `des_wyseur2007` is attacked using two possible target functions to show that different leakage models yield different results with CSR. The conditionally reduced results are for the first DES round only.

The `aes_nsc2013_variants` whitebox uses external encodings, which means normal DPA cannot recover the key. CSR still applies and can thus be used to reverse engineer the structure of the whitebox.

7 Case Studies

In this section, we compare the application of our approach (automated point selection and correlation-based differential attack, implemented in our tool) to the existing approach (optional manual range selection and correlation-based differential attack as implemented in the Daredevil tool [Sid]). Our targets are two whitebox implementations from the Deadpool collection: `des_wyseur2007` and `aes_ches2016`. We have chosen these two cases

for detailed studies, because they provide the longest traces (thus being more characteristic of real life examples), this makes the effect of our techniques more pronounced. We explain why and how our approach is superior to the existing one.

7.1 Methodology

We generate traces from whitebox implementations using either the tracer provided in the Deadpool repository as distributed in the Orka image, or our build of the tracer tool that allows us to dump more memory. In particular, this allows us to recover keys for `aes_ches2016`, where the original Deadpool approach does not (fully) recover the key.

We run Daredevil and our tool for increasing numbers of traces on a MacBook Air with a dual-core i7-5650U CPU, 8 GB RAM, and 512 GB SSD under macOS 10.13.2. We build Daredevil (commit `897f602`) natively on MacOS using MacPorts gcc6 6.4.0, and we run it with the maximum memory bound set to 6 gigabytes in the configuration file. We run AS-DPA with individual bits or (in the case of AES) models defined by Klemsa as target functions. Daredevil runs multi-threaded and in the case of DES performs only a single round attack, whereas our tool runs single-threaded and attacks two rounds of DES for the full key recovery.

We extract the following metrics from the logs of both tools:

- Runtime, to compare the efficiency of the tools
- Ranks of key byte candidates, to illustrate the effectiveness
- For our tool: the number of points remaining after application of point reduction methods, to explain the runtime behavior

We plot these metrics against the number of traces and interpret them in the following subsections.

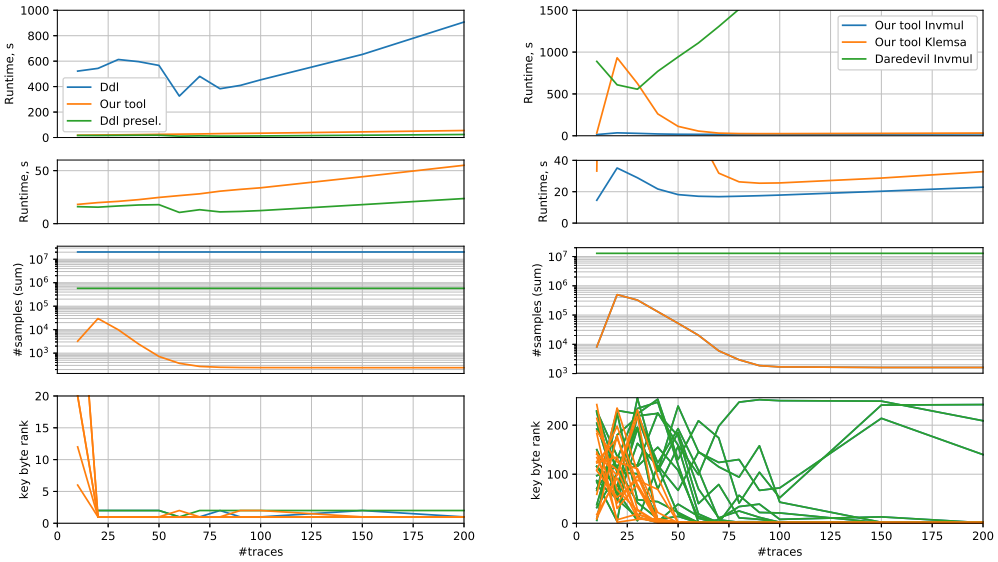
7.2 Case 1: `des_wyseur2007`

In this case, our approach lets us perform the attack without manual memory region selection, while keeping comparable runtime. The traceset size and required working memory are within the common laptop hard disk and RAM constraints, respectively.

The original DCA attack in Deadpool captures the traces of a short memory region that is preselected based on the manual inspection of memory plots, and runs Daredevil on these traces. This is our baseline. It is shown in Figure 2a as *Ddl presel.* For comparison, we have acquired a significantly larger traceset capturing a wide memory area (i.e. involving less analyst intuition) using our build of the tracer and run Daredevil and our tool on it for comparison. For Daredevil, *Ddl* in Figure 2a, such a large traceset leads to an order of magnitude longer runtime. With our tool, the automated point selection reduces the size of the trace and makes the runtime comparable to the baseline where the points are manually preselected, see the zoomed runtime subplot.

The plot with the number of samples shows the different data sizes that go into the correlation attack. For our tool, the number of points differs for different subkeys. Therefore, to allow for a comparison, we show the total number of samples. For our tool, it is the sum of the number of samples remaining after sample reduction for *all* the subkeys. Since there is no dimensionality reduction for Daredevil and for every subkey attack the original traces are used, the trace length needs to be multiplied by 8 to be comparable.

In this case, the Daredevil tool does not lead to the correct key chunk candidates being ranked on top, even with 200 traces, both in the traces of the preselected region and the larger traces. At the same time, our tool with automated point selection yields the top rank of the correct key byte candidates after 20 traces. This is most likely due to the



(a) `des_wyseur2007` (b) `aes_ches2016`

Figure 2: Different attack metrics vs number of traces. Top to bottom: runtime; zoomed-in runtime; total number of samples going into correlation computation; correct key chunk candidate ranks.

fact that our automated point selection leaves samples that are more appropriate for key recovery.

7.3 Case 2: `aes_ches2016`

In this case, our approach with sample reduction lets us perform the attack in a leakage model that recovers all key bytes at once significantly faster than the original attack, at the cost of additional traces. We perform the attacks on a traceset acquired with our build of the tracer.

The DCA attack in the Deadpool repository consists of two runs with different leakage models, S-box out and Invmul (the Rijndael multiplicative inverse in the S-Box), since none of the individual models leads to the recovery of all the subkeys. Here we perform the attack in the Invmul model with Daredevil and our tool. Though our approach takes significantly less time due to sample reduction, it cannot recover all the key bytes, since the leakage model is the same. Note that the key ranks obtained with our tool are exactly the same as for Daredevil (the green lines in the rank plot are on top of the blue lines in Figure 2b), because it is essentially the same attack.

For comparison, we perform the attack using our tool with all the 255 leakage targets described by Klemsa in [Kle16]. This approach needs significantly more computational effort than Invmul. However, with about 50 traces it recovers all the key bytes, and the number of samples left after our reduction techniques makes the runtime several times smaller than the shortest run of Daredevil (on 20 traces). With 75 traces the runtime of our tool becomes even smaller.

8 Conclusion

We describe novel techniques to reduce the amount of samples which go into a DCA attack when attacking whiteboxes. We show two techniques for automatically reducing the number of samples in the absence of noise: Duplicate Column Removal, and Conditional Sample Reduction. These methods reduce the computation time of the DCA attack. In the past, custom filters were used to reduce the amount of samples, but they rely on potentially faulty intuitions. In the Deadpool repository, such incorrect assumptions appear, namely in DCA on `aes_plaidctf2013`. The novel techniques can process more data, allowing effective but computationally intensive leakage models to be used, while having comparable runtime. For example, we apply the Klemsa leakage model on `aes_ches2016,` which allows us to recover the key.

Interestingly, with our methods the computation time can decrease with an increase in acquisitions. This is because point reduction methods can remove more samples when they have access to more information. This is particularly applicable in the DCA setting: binaries cannot fully enforce request limits. These methods allow the application of an automated and efficient analysis methodology: acquire more samples per trace and run the attack in less time.

Furthermore, we show differential plots which allow custom filters to be made with greater confidence, thereby reducing the amount of samples even further. As these differential plots are a generic whitebox reverse-engineering tool, they can make other whitebox analysis methods, such as DFA, more effective.

In general all described techniques reduce the dependency on analyst input and are a step forward for fully automated whitebox analysis.

References

[BB17] Paul Bottinelli and Joppe W. Bos. Computational aspects of correlation power analysis. *J. Cryptographic Engineering*, 7(3):167–181, 2017.

[BBIJ17] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, and Martin Bjerregaard Jepsen. Analysis of software countermeasures for whitebox encryption. *IACR Trans. Symmetric Cryptol.*, 2017(1):307–328, 2017.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, pages 16–29, 2004.

[BGE04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *SAC*, pages 227–240, 2004.

[BHMT16] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. *CHES*, page 215, 2016.

[CEJVO02] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot. White-box cryptography and an AES implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[DGLHL13] 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*, pages 839–850. ACM, 2013.

[DPRS11] Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate side channel attacks and leakage modeling. *J. Cryptographic Engineering*, 1(2):123–144, 2011.

[Exp] Repository with tracesets and scripts. https://github.com/ikizhvatov/conditional-reduction.

[JBF02] Matthias Jacob, Dan Boneh, and Edward W. Felten. Attacking an obfuscated cipher by injecting faults. In *DRM*, pages 16–31, 2002.

[Jls] Jlsca: Side-channel toolkit in Julia. https://github.com/Riscure/Jlsca.

[JR02] Markus Jakobsson and Michael K. Reiter. Discouraging software piracy using software aging. In *ACM CCS Workshop DRM*, pages 1–12. Springer, 2002.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[Kle16] Jakub Klemsa. Side-channel attack analysis of AES white-box schemes. Master's thesis, Czech Technical University in Prague, 2015/2016. https://github.com/fakub/DiplomaThesis.

[LPR13] Victor Lomné, Emmanuel Prouff, and Thomas Roche. Behind the scene of side channel attacks. In *ASIACRYPT*, pages 506–525. Springer, 2013.

[pin] Pin – a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[Sid] Side-Channel Marvels. https://github.com/SideChannelMarvels.

[SMdH] Eloi Sanfelix, Cristofaro Mune, and Job de Haas. Unboxing the white-box. Black Hat Europe 2015.

[Teu15] Philippe Teuwen. MoVfuscator writeup. http://wiki.yobi.be/wiki/MoVfuscator_Writeup#Attack, 2015.

[val] Valgrind. http://valgrind.org/.

[whi17] CHES 2017 capture the flag challenge: The WhibOx contest. https://whibox.cr.yp.to, 2017.

[WO11] Carolyn Whitnall and Elisabeth Oswald. A fair evaluation framework for comparing side-channel distinguishers. *J. Cryptographic Engineering*, 1(2):145–160, 2011.

A Step-by-Step Example for CSR

Here we show how the CSR algorithm described in section 4 applies to the case of DCA on DES with S-box output as a target. Namely, the target function f_1 in this attack is $ob_1 = f_1(kb_1, ib_1) = \text{Sbox}_1[kb_1 \oplus ib_1]$ where we will guess a 6-bit key chunk kb_1, we know 6-bit input ib_1 and predict a 4-bit output ob_1. Target function input ib_1 is computed only from the DES 8-byte input or output: some bit permutations are applied, but no secret is mixed in. The DES S-box attack has 8 target functions, one for each S-box.

Suppose we made 3 memory recordings of a DES execution, all with the same key, and we recorded the cipher inputs. Table 3 shows the cipher inputs used, corresponding target inputs ib_1, and the observations, in bits.

Table 3: Example DES whitebox observations

row	cipher input (hex)	ib_1 (hex)	O (bits)
1	`0xf537eed56d776fdd`	`0x11`	`10100101`
2	`0x7c0bbce8435838cd`	`0x11`	`10001110`
3	`0x45e126942f699d75`	`0x09`	`11010000`
4	`0x1439f7d4510329d3`	`0x11`	`10010000`

We have 3 observations where target input ib_1 is `0x11`. If the target function input is the same, the target function output will be the same too. Under this assumption, for target function f_1, we can discard all the columns c in O were $O_{r,c}$ are different between rows $r = 1$, $r = 2$ and $r = 4$. CSR computes mask M where element $M_{k,c}$ is set to 1 if column c in all observations for target k do not differ between different observations with the same target input. We run CSR for target f_1.

We invoke CSR (Algorithm 2) as follows.

$n = 4$

$m = 8$

O as defined in Table 3

D as defined by column ib_1 in Table 3

Then, initialize M to `11111111`

Initialize C to $\emptyset$

After iteration 1:

$M = \texttt{11111111} \wedge (\neg(\texttt{10100101} \oplus \texttt{10100101}) = \texttt{11111111}$

$C = \{\texttt{0x11} : 1\}$

After iteration 2:

$M = \texttt{11111111} \wedge (\neg(\texttt{10001110} \oplus \texttt{10100101}) = \texttt{11010100}$

$C = \{\texttt{0x11} : 1\}$

After iteration 3:

$M = \texttt{11010100} \wedge (\neg(\texttt{11010000} \oplus \texttt{11010000}) = \texttt{11010100}$

$C = \{\texttt{0x11} : 1, \texttt{0x9} : 3\}$

After iteration 4:

$M = \texttt{11010100} \wedge (\neg(\texttt{10010000} \oplus \texttt{10100101}) = \texttt{11000000}$

$C = \{\texttt{0x11} : 1, \texttt{0x9} : 3\}$

Return:

$n' = 2$

$m' = 2$

$X' = \{\texttt{0x11}, \texttt{0x9}\}$

$O' = \left\{ \begin{array}{c} \texttt{10} \\ \texttt{11} \end{array} \right\}$

That is, CSR reduced the number of inputs from 4 to 2 (unique ones for this target function), and trace length from 8 bits to 2 bits.