CLAASP: a Cryptographic Library for the Automated Analysis of Symmetric Primitives

Emanuele Bellini, David Gerault, Juan Grados, Yun Ju Huang, Mohamed Rachidi, and Sharwan Tiwari

Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
{emanuele.bellini,david.gerault,juan.grados,yunju.huang,
mohamed.rachidi,sharwan.tiwari}@tii.ae

Abstract. This paper introduces CLAASP, a Cryptographic Library for the Automated Analysis of Symmetric Primitives. The library is designed to be modular, extendable, easy to use, generic, efficient and *fully* automated. It is an extensive toolbox gathering state-of-the-art techniques aimed at simplifying the manual tasks of symmetric primitive designers and analysts. CLAASP is built on top of Sagemath and is open-source under the GPLv3 license.

The central input of CLAASP is the description of a cryptographic primitive as a list of connected components in the form a directed acyclic graph. From this representation, the library can automatically: (1) generate the Python or C code of the primitive evaluation function, (2) execute a wide range of statistical and avalanche tests on the primitive, (3) generate SAT, SMT, CP and MILP models to search, for example, differential and linear trails, (4) measure algebraic properties of the primitive, (5) test neural-based distinguishers.

In this work, we also present a comprehensive survey and comparison of other software libraries aiming at similar goals as CLAASP.

Keywords: Cryptographic library · Automated analysis · Symmetric primitives

1 Introduction

The security targets for cryptographic primitives are well-defined, and relatively stable, after decades of cryptanalysis. In particular, a symmetric cipher should behave like a random keyed permutation, a hash function should behave like a random function, and a MAC scheme should be unforgeable. Testing a cryptographic primitive for these properties is, on the other hand, a vastly difficult task that relies on testing for known weaknesses. Such a process generally involves determining the most likely differential or linear characteristic, evaluating the resistance of the primitive to various cryptanalysis techniques such as integral attacks, and running generic randomness tests. Fortunately, automatic techniques exist to help designers and cryptographers run such evaluations; for instance,

SAT/SMT, Mixed Integer Linear Programming (MILP) or Constraint Programming (CP) are frequently used to find optimal differential and linear characteristics. These tools have, over time, become more accessible to non-experts, through libraries such as [52], that generate models (in this case, SMT) automatically from a description of the cipher. However, such tools generally focus on a single aspect, such as generating models in a given paradigm, and there is currently no single-stop toolkit that combines automated model generation, statistical testing and machine learning based analysis. We aim to fill this gap with CLAASP, a Cryptographic Library for the Automated Analysis of Symmetric Primitives. This paper introduces the first public version of CLAASP; the ambition of the project is to keep adding analysis tools in line with the state of the art, to provide cryptanalysts with a click-of-a-button solution to run all the standard analysis tools and gain an overview of the security of a given primitive.

We first present existing cryptanalysis libraries in Section 1.1, before introducing the building blocks of CLAASP: the cipher object in Section 2, and the evaluators in Section 3. We then present the battery of tests and tools implemented in CLAASP in Section 4, and finish with a comparison with other cryptographic libraries in Section 5.

The library's source code has been made available to the wider community and is publicly accessible at (Github: https://github.com/Crypto-TII/claasp). Also, in (Github: https://github.com/peacker/claasp_white_paper), you can find the scripts used to accompany this paper.

1.1 Related works

Automated tools to support cryptanalysts have become a cornerstone for the design of new primitives. Over time, such tools were made more generic and gathered into libraries; we describe the most prominent ones in this section.

The lineartrails library [30] is dedicated to the search for linear characteristics on SPN ciphers. ARX toolkit [42,43] and YAARX [63] focus on ARX ciphers, the former testing conditions for trails to be possible, and the latter performing various analysis techniques on the components.

On the algebraic cryptanalysis side, the Automated Algebraic Cryptanalysis tool [60] tests properties of block and stream ciphers; in particular, it evaluates the randomness of a cipher through Maximum Degree Monomial tests [61].

Autoguess [37] is a tool to automate the technique guess-and-determine. This technique involves making a calculated guess of a subset of the unknown variables, which enables the deduction of the remaining unknowns using the information obtained from the guessed variables and some given relations. In order to automate this technique, SAT/SMT, MILP, and Gröbner basis solvers are used and several new modeling techniques to exploit these solver proposed. For instance, the authors of the library introduce new encodings in CP and SAT/SMT to solve the problem of determining the minimal guess, i.e., the subset of guessed variables from which the remaining variables can be deduced. Autoguess also allows to automate the key-bridging technique. This technique is utilized in key-recovery attacks on block ciphers, wherein the attacker seeks to determine the

minimum number of sub-key guesses needed to deduce all the involved sub-keys through the key schedule. The significant contribution of this work lies in integrating key-bridging techniques into tools that were previously only capable of searching for distinguishers. As a result, these enhanced tools can now be utilized as fully automatic methods for recovering keys.

CryptoSMT [62] is the first large-scale solver-based library dedicated to cryptanalysis. Based on SMT and SAT solvers, it provides an extensive toolkit, permitting the search for optimal differential and linear trails, the evaluation of the probability of a differential, the search for hash function preimages, and secret key search.

Another SMT-based library, based on ArxPy [52] is the CASCADA framework [53], which also implements techniques to search for rotational-XOR differentials, impossible-rotational-XOR, but also related-key impossible-differentials, linear approximations, and zero-correlation characteristics. The generated SMT models are expressed through the theory of bit-vectors [8], and follow the general methodology of Mouha and Preneel [47] for differential properties, Sasaki's [55] technique for impossible differentials, an SMT-based miss-in-the middle search for related-key impossible differentials of ARX ciphers [5], and a novel method proposed for zero-probability global properties. If a search can not use the previous methods, then a generic method, based on the constructions of statistical tables, such as the Differential Distribution Table (DDT), is used. Depending on the sizes of the inputs of the block cipher, these generic models could be costly, so they also proposed heuristic models by relaxing the accuracy of their properties; they called them weak models. Finally, their framework implements methods to check the properties mentioned above experimentally.

Finally, TAGADA [44] is a tool which generates Minizinc [50] models for the search for differential properties on word-based SPN ciphers, such as the AES. The search for such ciphers is typically divided into two steps, one where the word variables are abstracted as boolean values denoting the presence or absence of a difference, and one where the abstracted solutions from step 1 are instantiated to word values, when possible. The models generated by TAGADA implement the first step, including optimisations based on inferred equalities through XOR operators, in order to drastically reduce the number of incorrect solutions to be passed to step 2. Such constraints are deduced naturally from a Directed Acyclic Graph (DAG) representation of the cipher under study. The genericity of Minizinc models enables solving with a range of CP, SAT and SMT solvers, in particular, the ones participating in the MiniZinc competition, that provide an interface to MiniZinc. On the other hand, solver-specific optimisations and perks are abstracted away by the Minizinc interface, compared to models developped in the native language of a solver.

A summary of the functionalities of these libraries is presented in Table 1.

1.2 Our contribution

We introduce CLAASP, a Cryptographic Library for the Automated Analysis of Symmetric Primitives. CLAASP has been designed to simplify the manual

		TAGADA	CASCADA	CryptoSMT	lineartrails	YAARX	Autoguess	CLAASP
Cipher types		SPN	All	All	SPN	ARX	All	All
Cipher representation		DAG	Python code	Python code	C++ code	C code	Algebraic representation	DAG
Statistical/Avalanche tests		-	-	-	-	-	-	Yes
Continuous diffusion tests		-	-	-	-	-	-	Yes
Components analysis tests		-	-	-	-	-	-	Yes
Constraint solvers	Differential trails	Truncated	Yes	Yes	-	Yes	-	Yes
	Differentials	-	Yes	Yes	-	Yes	-	Yes
	Impossible differential	-	Yes	-*	-	-	-	Yes
	Linear trails	-	Yes	Yes	Yes	-	-	Yes
	Linear hull	-	-*	-*	-	-	-	Yes
	Zero correlation approximation	-	Yes	-*	-	-	-	Yes
	Supported solvers	CP, (MiniZinc)	SMT	SMT	-	-	SAT/SMT, MILP,CP Groebner basis	SAT, SMT, MILP, CP, Groebner basis
	Supported Scenarios	single-key related-key	single-key related-key	single-key related-key	single-key	single-key	single-key related-key single-tweak related-tweak	single-key related-key single-tweak related-tweak
Algebraic tests		-	-	-	-	-	-	Yes (algebraic model for cipher preimages)
Neural-based tests		-	-	-	-	-	-	Yes
State Recovery		-	-	-	-	-	Yes	-
Key-bridging		-	-	-	-	-	Yes	-

Table 1: Comparison of cryptanalysis libraries features with CLAASP. -* means that the functionality is not supported, but could easily be added from the existing code.

tasks of symmetric cipher designers and analysts. CLAASP has been designed with the following goals:

- Be *open-source* with a GPLv3 licence.
- Be *modular*. For this reason it is built on top of Sagemath, thus inheriting Python modularity.
- Be *extendable*. The Python/Sagemath environment allows to easily integrate other powerful libraries: constraint solvers such as Cryptominisat, Cadical or Gurobi, machine learning engines such as Tensorflow, Grobner basis solvers, parallelization packages such as NumPy, etc..
- Be *usable*. Much effort has been dedicated to provide a smooth user experience for both designing and analyzing a cipher. This includes a comprehensive documentation for users and developers, and a Docker image to easily start with the library without the need of installing all the dependencies.
- Be *generic*. The wide range of pre-defined components, allows to implement a wide range of iterated symmetric ciphers, ranging from block ciphers (possibly with a tweak), cryptographic permutations, hash functions, and covering several design types such as Feistel, SPN, ARX, etc..
- Be *automated*. The concept of the library revolves around providing a cipher design as the input and getting an analysis of the cipher design as the output with respect to some desired property.

– Be *efficient*. In spite of being the most generic and fully automated tool of its kind, this library is competitive in terms of efficiency with similar tools targeting specific sectors.

The central objects of CLAASP are symmetric ciphers. They are described as directed acyclic graphs whose nodes are components (S-Boxes, linear layers, constants, Input/Output, etc.) and whose edges are input/output component connections. From this representation, the library can automatically:

1. generate the Python or C code of the evaluation function;
2. execute a wide range of statistical and avalanche tests on the primitive, including continuous diffusion tests;
3. generate a report containing the main properties of the cipher components (e.g. S-Box differential uniformity or algebraic degree, linear layer order or branch number, etc.);
4. generate SAT, SMT, CP and MILP models and feed them to most open-source and commercial solvers, in order to search, for example, differential and linear trails;
5. measure algebraic properties of the primitive;
6. test neural-based distinguishers.

Beside the presentation of the library, important contributions of this work are a survey and a comparison (where possible) of the main software tools trying to achieve the same goals as CLAASP.

2 Symmetric primitives in CLAASP

In this section, we describe how a symmetric primitive is represented in CLAASP. We also present the main pre-implemented primitives that are available for testing and give some indications on how to build a custom cipher.

2.1 The Component class

Informally, in CLAASP, a symmetric cipher is represented as a list of "connected components". By the term *cipher component* (or simply *component*) we refer to the building blocks of symmetric ciphers (S-Boxes, linear layers, word operations, etc.). Two components are *connected* when the output bits of the first component become the input bits of the second component, in a one-to-one correspondence. The library supports the following *primitive* components: the S-Box component, linear layer components (fixed and variable rotation, fixed and variable shift, bit and word permutation, multiplication by a binary or word matrix), word operations components (NOT, AND, OR, XOR, modular addition and subtraction), and the constant component. It also supports *composite* components, which are a combination of primitive components: the sigma function used in ASCON, the theta function used in Keccak, and the theta function used in Xoodoo. For example, the linear layer in ASCON can be presented by

the combination of several XOR and ROTATE components, or as a composite component. Composite components can also be created at a user level.

Finally, some special components are used to represent the inputs of the cipher, and cipher intermediate and final outputs.

In CLAASP, each component requires the following minimal information to be defined:

- a unique component ID (e.g. "sbox_0_0");
- a component type (e.g. "sbox", "word_operation", "linear_layer", etc.);
- the input and output bit size of the component;
- a list of the components that are connected to the input of the component (a list of IDs);
- a list of lists of bits positions specifying which output bits of the input components are connected to the component;
- a description containing the necessary information to finalize the definition of the component (e.g., the list of integers defining an SBox, the binary matrix defining a linear layer, the amount of a rotation, etc.).

More precisely, in CLAASP, a component is represented as a Python class with the following constructor:

```
def __init__(self, component_id, component_type, component_input,
    ↪ output_bit_size, description):
```

The *component input* is represented as another class defined by the following constructor:

```
def __init__(self, input_bit_size, id_links, bit_positions):
```

2.2 The Cipher class

Ciphers as directed acyclic graphs In CLAASP, a symmetric cipher is represented as a list of connected components, forming a directed acyclic graph, and a list of basic properties, listed in Table 2.

Property	Description
id	unique identifier of the cipher, composed by cipher name and parameters
family_name	name of the cipher family, such as AES ASCON, etc.
type	type of the cipher (block cipher, permutation, hash or stream cipher)
inputs	inputs of the cipher, such as key and plaintext.
inputs_bit_size	list of number of bits of each input parameters.
output_bit_size	number of bits of the cipher output
number_of_rounds	number of rounds in the cipher
rounds	list of rounds each containing a list of components
reference_code	[optional] Python reference code (as a string) of the cipher evaluation function, used to verify the cipher correctness.

Table 2: Parameters that are used to define a cipher in CLAASP.

CLAASP supports *iterated symmetric ciphers*, based on the composition of several round functions, which are themselves a list of connected components;

each cipher must have at least one round. The round decomposition is useful and common in symmetric cipher design and cryptanalysis; in most tests, a given property is studied round by round.

CLAASP natively implements a range of well-known block ciphers, permutations and hash functions, listed in Table 3.

Block ciphers		Permutations		Hash functions
AES [59]	TEA [64]	ASCON [31]	Xoodoo [27]	SHA-1
DES [49]	XTEA [65]	ChaCha [17]	Spongent-π [22]	SHA-2
LEA [39]	Twofish [56]	GIFT-128 [7]	TinyJAMBU [66]	MD5
LowMC [1]	Threefish [32]	GIMILI [18]		BLAKE [4]
Midori [6]	HIGHT [40]	Grain core [38]		BLAKE2 [4]
PRESENT [21]	SKINNY [13]	KECCAK-p [19]		
Raiden [51]	Sparx [29]	PHOTON [36]		
SIMON [11]	Speck [11]	SPARKLE [12]		

Table 3: Primitives supported in CLAASP v1.0.0.

How to create the cipher object Native support for more primitives will be added over time, but CLAASP exposes a simple interface for users to add new ones as well. This process is illustrated through a toy example of a 2-rounds cipher with 6-bit block, 6-bit key injected in every round with a XOR operation, 2 3-bit S-boxes, and a linear layer made of a left rotation of 1 bit, shown in Figure 2, and the corresponding CLAASP implementation in Figure 1.

The main concern of a user implementing a primitive is to correctly link the components at a bit level, and mark which component or group of components need to be reported in the output of the tests. This is because a user might be interested not only in getting reports at every round, but, for example, after the linear and the nonlinear layer of an SPN.

Cipher inputs It is important to notice that, in order to be generic, the library has been designed to accept multiple inputs which can be labeled with different names: for example, a key, a plaintext and a tweak, or a message and a nonce. On the other hand, to better exploit the features of some tests, a naming convention has been introduced for inputs such as "key" or "plaintext".

The *cipher representation* is not unique The *cipher representation* as a list of connected components is *not* unique. For example, the nonlinear layer of ASCON permutation can be represented as a circuit made of word operation components (XOR, AND and NOT) or with a layer of parallel S-boxes. This is detailed in Appendix A.

Different *cipher representations* may affect the output of tests; for instance, a differential cryptanalysis model built for an ASCON implementation using the circuit representation is less accurate than one using a S-Box representation. In

```
from claasp.cipher import Cipher

class ToySPN(Cipher):
  def __init__(self):
    super().__init__(family_name="toyspn",
      cipher_type="block_cipher",
      cipher_inputs=["plaintext", "key"],
      cipher_inputs_bit_size=[6, 6],
      cipher_output_bit_size=6)

    sbox = [0, 5, 3, 2, 6, 1, 4, 7]
    self.add_round()
    xor = self.add_XOR_component(["plaintext", "key"
          ↪ ],[[0,1,2,3,4,5],[0,1,2,3,4,5]],6)
    sbox1 = self.add_SBOX_component([xor.id], [[0, 1,
          ↪ 2]], 3, sbox)
    sbox2 = self.add_SBOX_component([xor.id], [[3, 4,
          ↪ 5]], 3, sbox)
    rotate = self.add_rotate_component([sbox1.id,
          ↪ sbox2.id],[[0, 1, 2], [0, 1, 2]], 6, 1)
    self.add_round_output_component([rotate.id], [[0,
          ↪ 1, 2, 3, 4, 5]], 6)

    self.add_round()
    xor = self.add_XOR_component([rotate.id, "key"
          ↪ ],[[0,1,2,3,4,5],[0,1,2,3,4,5]],6)
    sbox1 = self.add_SBOX_component([xor.id], [[0, 1,
          ↪ 2]], 3, sbox)
    sbox2 = self.add_SBOX_component([xor.id], [[3, 4,
          ↪ 5]], 3, sbox)
    rotate = self.add_rotate_component([sbox1.id,
          ↪ sbox2.id],[[0, 1, 2], [0, 1, 2]], 6, 1)
    self.add_cipher_output_component([rotate.id], [[0,
          ↪  1, 2, 3, 4, 5]], 6)

toyspn = ToySPN()
hex(toyspn.evaluate([0x3F,0x3F]))
```

Fig. 1: ToySPN class definition.

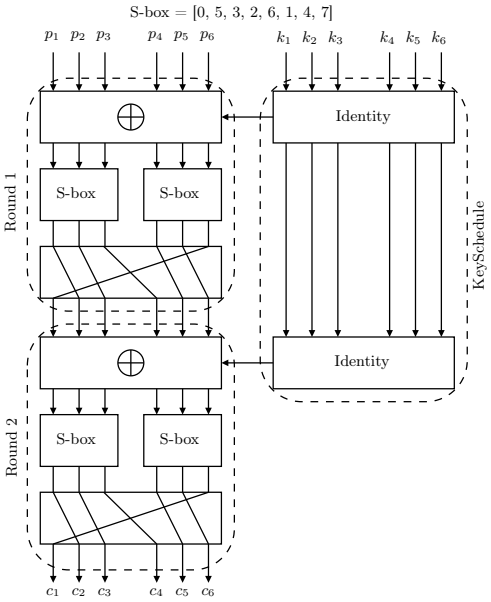


Fig. 2: ToySPN diagram.

general, the circuit model is useful when a user wishes to monitor the action of every gate (i.e. word operation) on a single bit. On the other hand an S-Box-based model often allows a faster evaluation function, and more precise automated search for differential and linear trails for some constraint solvers such as CP, where the search can have a preliminary filter to identify all possible active S-Boxes configurations. Another example of different representation is the use of binary matrices as opposed to word-based matrices in linear layers.

To test the properties mentioned above, CLAASP already contains some primitives with both circuit and S-Box-based representation, such as ASCON, Xoodoo, Keccak and Gimli, as well as the bit-based and word-based such as TinyJambu representations.

3 Library: evaluation modules

The most basic functionality of CLAASP is to evaluate a cryptographic primitive on a given input. This can be easily achieved in few lines of Python code. However, some statistical tests require the evaluation of millions of inputs, and looping over all inputs is not practical, due to Python's well-known sluggishness with loops. In CLAASP, this issue is tackled through different options, namely vectorized implementations, and C code generation. A further speedup, to appear in future versions, is CUDA-based parallel evaluation with GPUs.

3.1 Base Evaluator in Python and C

One essential functionality for a cryptographic primitive is being able to evaluate it over some input. In CLAASP, users can create a cipher object and call an evaluation method to evaluate a particular input. This functionality is also used internally in CLAASP by some of the modules, to run, for example, avalanche or statistical tests. By inserting output components in the cipher, users may also intercept and visualize the intermediate output of any desired component or group of components during the evaluation. Both the Python code or the C code[1] to evaluate a cipher are generated automatically by scanning the list of the cipher components, generating the corresponding block of code and linking each block in the correct order. The automatically generated code is not optimized. However, it provides an easy way for users to export the code for quick prototyping. The optimization of the automatically generated code is planned for future versions of the library.

3.2 Vectorized Implementations

A *vectorized* implementation of a function handles multiple inputs, presented as a vector, at the same time. In Python, the NumPy library allows to parallelize function evaluations, by running the function on an *array* of inputs, rather than a single input. NumPy arrays are typed and homogeneous, which, combined with NumPy's optimisations, enables significant performance gains compared to Python native lists.

The cipher object provides the NumPy-based *evaluate_vectorized* function, which can be used for the fast evaluation of an array of inputs. The inputs are specified as NumPy arrays, of 8-bit unsigned integer values, arranged as one column per data point. The return value is encoded as a list containing a single NumPy array of 8-bit unsigned integer values, this time arranged as one row per data point. The choice of using bytes stems from NumPy's lack of support for integers over 64 bits, and the ease to generate such format automatically.

3.3 Performance Evaluation

For examples of how to easily use the Python, C and vectorized evaluation method, please refer to the script `evaluation_benchmark.sage` in the repository accompanying this publication. The performance of the primitives' evaluators are compared in Table 4. Note that since the code are auto-generated and not optimized, the table does not indicate the efficiency of the specified primitives. Note that single evaluation in NumPy is usually faster than the single evaluation using Python or even C. Yet, it is convenient to keep Python and C for very few evaluations as the input/output format is more intuitive as it is represented by an integer. Finally note that the time reported for C also include the time to compile the C program.

[1] When possible a word-oriented implementation is used, opposed to a slower bit-oriented implementation for primitives with mixed type of components.

	block size	round	Python		C		Vectorized		
			1	10^3	1	10^3	1	10^3	10^6
SKINNY	128	40	4.32	3546.98	2.87	1545.29	1.22	1.10	14.27
AES	128	10	0.80	739.26	1.59	765.86	0.27	0.28	2.79
HIGHT	64	32	0.83	848.06	1.53	627.53	0.11	0.22	1.33
LEA	128	24	1.51	1391.93	1.70	771.62	0.08	0.08	4.77
LowMC	128	20	3.05	2922.92	2.50	1710.59	1.80	2.24	907.42
Midori	128	20	1.53	2093.48	2.24	1204.19	0.64	0.80	105.55
SIMON	128	68	3.19	3163.11	1.37	755.87	0.09	0.10	8.18
Speck	128	32	1.46	1467.64	0.95	432.67	0.05	0.06	6.09
Raiden	64	16	0.78	770.91	0.94	433.65	0.05	0.08	7.75
Sparx	128	8	1.68	1726.98	1.24	810.48	0.22	0.24	5.89
TEA	64	32	1.12	1127.31	0.99	439.49	0.09	0.09	8.66
XTEA	64	32	1.00	1052.29	0.94	443.84	0.06	0.07	7.20
Threefish	256	72	3.76	3883.64	0.84	778.91	0.19	0.19	29.57
ASCON	320	12	3.05	2050.94	7.23	416.29	0.17	0.07	4.25
Gift	128	40	1.85	1565.64	1.40	799.74	0.17	0.18	8.38
Keccak	200	18	2.20	1989.07	1.63	605.79	0.26	0.24	2.80
PHOTON	256	12	1.18	942.26	1.28	703.64	0.31	0.28	22.46
Spongent-π	160	80	7.77	7916.27	3.97	3715.62	5.07	6.80	2300.32
TinyJAMBU	128	32	0.43	411.65	1.02	533.11	0.08	0.07	3.51
Xoodoo	384	12	2.06	2096.78	1.23	701.80	0.20	0.20	4.32
SPARKLE	256	10	1.75	1874.37	1.38	780.80	0.09	0.09	6.05
GIMILI	384	24	3.31	3053.50	1.03	558.23	0.17	0.16	7.05
Grain core	80	160	0.93	909.32	1.57	847.19	0.23	0.22	11.03
ChaCha	512	20	1.19	1144.58	1.19	517.94	0.06	0.07	5.42
SHA-1	160	80	2.14	1926.06	1.34	418.85	0.12	0.13	10.45
SHA-2	256	65	4.20	4515.25	0.97	545.93	0.18	0.20	20.68
MD5	64	64	1.36	1453.34	1.11	610.73	0.08	0.09	7.27
BLAKE	512	28	5.26	4651.17	1.62	545.44	0.32	0.31	22.79
BLAKE2	1024	12	5.49	5719.36	0.90	528.41	0.27	0.32	39.54

Table 4: Primitives evaluator performance in CLAASP with 1, 10^3 and 10^6 inputs. The timings are in seconds.

4 Library: test modules

In this section, we describe all automated analysis modules that are currently supported in CLAASP.

4.1 Component analysis

This module allows the visualization of the "quality" of certain properties of the components used in a cipher, by means of radar charts. These properties include:

- Boolean function properties, such as number of terms, algebraic degree, number of variables, wheter the Boolean function is APN or balanced;
- vectorial Boolean function properties such as differential uniformity, boomerang uniformity, nonlinearity, etc.;
- linear layer properties such as order, linear and differential branch number.

An example of 2 rounds AES-128 is illustrated in the method `component_analysis_tests` of the library. This method outputs a list of dictionaries, each of them containing information about an operation of the cipher. In this particular example, the first element of this list contains the following:

```
{'type': 'word_operation',
 'input_bit_size': 256,
 'output_bit_size': 128,
 'description': ['XOR', 2],
 'number_of_occurrences': 3,
 'component_id_list': ['xor_0_0', 'xor_0_36', 'xor_1_31'],
 'properties': {'degree': {'value': 1.0,
   'min_possible_value': 1,
   'max_possible_value': 256},
  'nterms': {'value': 2.0, 'min_possible_value': 1, 'max_possible_value': 2},
  'nvariables': {'value': 2.0,
   'min_possible_value': 1,
   'max_possible_value': 256}}}
```


We can see that it corresponds to a XOR operation between 2 inputs of 128 bits. This operation occurs 3 times in the 2 rounds AES and the IDs of these occurrences are also reported. Other properties we can observe are the algebraic degree of the component output bits expressed as a Boolean function, the number of terms, and the number of variables, respectively $1, 2, 2$.

For a better visualization, this module can also plot the results of the observation in a radar chart, such as the one presented in Figure 3 for the sbox of AES A full list of the radar charts of the components of 2 rounds of AES is given in the Appendix C. The method to generate these charts is `print_component_analysis_as_radar_charts`.

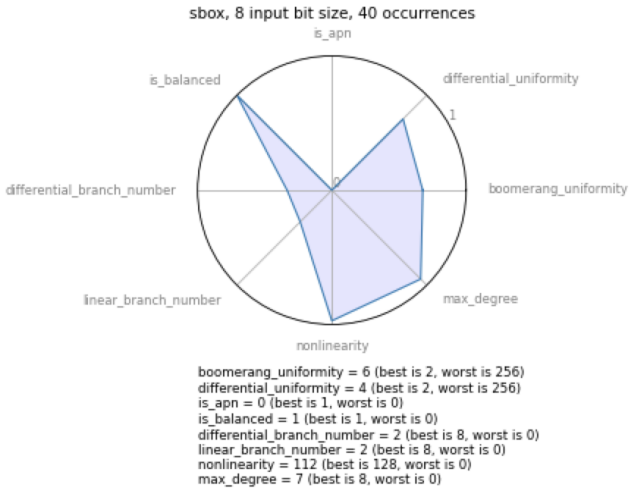


Fig. 3: Observation of the open-source of AES s a radar chart

4.2 Statistical and avalanche tests

Statistical tests Statistical tests aim at evaluating the randomness of a set of bit strings. Such tests were applied to evaluate AES candidates [57,58,10] through the NIST Statistical Test Suite (NIST STS) [54,9]. In addition, tools such as Diehard [45], or its successor Dieharder [24], provide additional statistical tests. CLAASP integrates both

the NIST STS and Dieharder suites within the statistical test module. The statistical test process is divided into two phases, dataset generation and analysis, as shown in Figure 4.

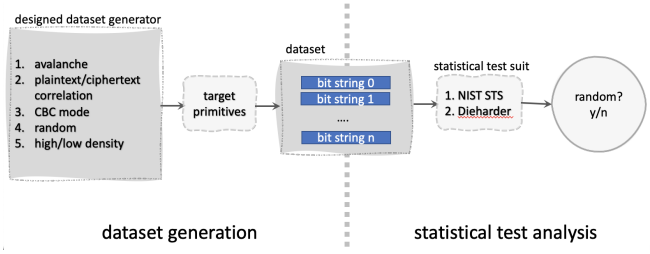


Fig. 4: Illustration of the process of statistical test.

Dataset generator The predefined datasets of CLAASP are defined in [57], which only covers keyed primitives. Keyless primitives datasets are some how special cases of the keyed ones. As an example, the illustration of the avalanche dataset generator is shown in Appendix B. For details of other dataset generators, please refer to [57]. The dataset generator is based on CLAASP's vectorized evaluation method, which returns a set of bit strings, as shown in Figure 4.

Statistical test tools The use of NIST STS and Dieharder in CLAASP is illustrated in the `run_statistical_test.sage` script. The results are exported as a report, and additionally returned as a Python dictionary for easy integration. CLAASP also features visualization of the results, as shown in Figure 5.

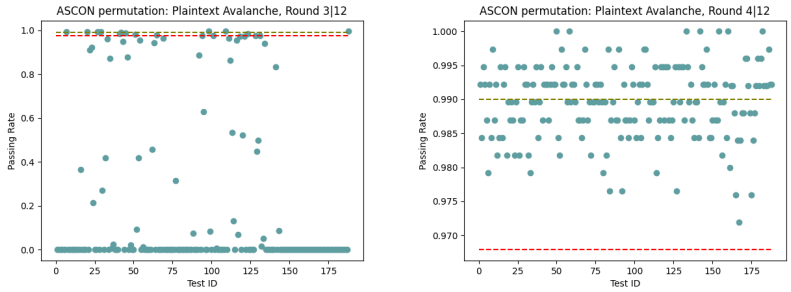


Fig. 5: CLAASP plot for the 188 NIST statistical tests pass rate of ASCON round 3 and round 4.

Performance and experiments To generate the plaintext avalanche test for all supported primitives (191 Gigabits), it takes 4 hours. For a 100 Mbits dataset, it takes

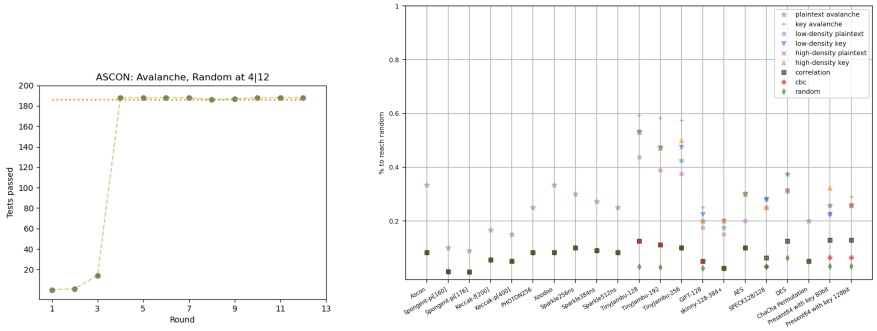


Fig. 6: Randomness graphs of ASCON generated by CLAASP. Left side is the statistical test result of avalanche dataset. Right side are all the statistical results of ASCON compared with other primitives.

around 30 minutes to finish the NIST statistical tests. Figure 6 shows the number of tests that pass for each round of ASCON (left) and the percentage of the rounds needed to pass all statistical tests with respect to the 9 possible datasets for several primitives.

Avalanche tests This module focuses on the avalanche properties, presented in [27], of a symmetric iterated primitive. These tests evaluate the cipher with respect to three different metrics that represent what usually the literature call *full diffusion*, *avalanche* and *strict avalanche* criteria. The goal of the tests is to compare how these metric evolve with respect to the computational cost of the round function; each metric is expected to satisfy a certain criterion (namely to pass a threshold) after a few rounds.

Usage. CLAASP exposes the *diffusion_tests* method for avalanche tests. The results are returned as a dictionary, from which a user can:

- check if a criterion is satisfied at a certain round for a specific input bit difference.
- obtain the worst input bit differences, that are the input differences for which the criterion is satisfied the latest.
- obtain the value of the criterion for a specific round and a specific input bit difference.
- obtain the average value of the criterion among all the input bit differences for a specific round.

For a better visualization, CLAASP can generate a heatmap graph from the dictionary returned by the avalanche tests using the method *generate_heatmap_graphs_for_avalanches_tests*. This is illustrated for 5 rounds of ASCON320 in Appendix D, which represents the heatmap graphs for the entropy criterion when the input bit difference has been injected in position 0. Each cell of this figure is greener if the entropy based on the probability of flipping of the underlying bit is close to 1, with a darker shade of red otherwise.

Timings. Figure 7 reports the timings of the avalanche tests for 5 rounds of some popular ciphers, using the vectorized evaluation function, up to 50,000 samples; all tests run globally in less than 5 minutes.

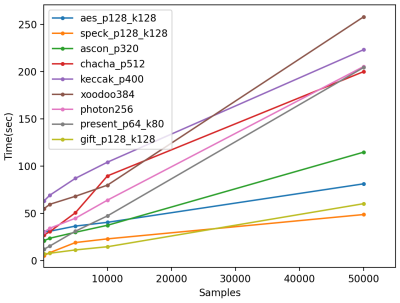


Fig. 7: Timings of the avalanche tests for five rounds of popular ciphers

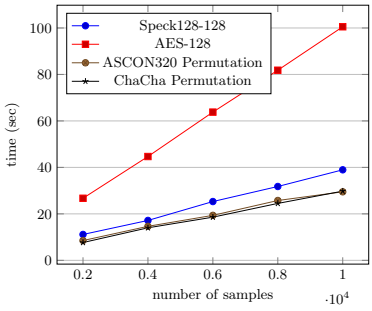


Fig. 8: Time comparison of the CAF computation for Speck128-128, AES-128, the iterated permutation in ASCON320 cipher, and the iterated permutation in ChaCha cipher, fixed to 5 rounds each for several random samples.

4.3 Continuous diffusion tests

In [25], Coutinho *et. al*, describe a framework to construct continuous functions from Boolean ones. Assuming independence, these functions provide the probability or correlation between the output bits being 1 based on an input of real numbers that represent the probability of each input bit being 1. For example, the authors construct the continuous version of the AND operator ($\odot$): Suppose we want to compute $p_3 = \Pr[a \odot b = 1]$, where a and $b \in \mathbb{F}_2$ are independent random variables. If $\Pr[a = 1] = p_1$ and $\Pr[b = 1] = p_2$, then $p_3 = p_1 p_2$. By using this expression, they define a continuous operator from $\odot$, called "continuous generalization of $\odot$". Specifically, they provide the definitions using the correlation of the random variables instead of probabilities. More precisely, let $\Pr(E)$ be the probability of occurrence of an event E and $b \in \mathbb{F}_2$ be a bit, then we can write $\Pr(b = 1)$ in terms of its correlation ϵ as $\Pr(b = 1) = p = \frac{1}{2}(1+\epsilon)$. In our example, expressing p_1, p_2 and p_3 as functions of their correlations, we have $p_1 = \frac{1}{2} + \frac{\epsilon_{p_1}}{2}$ and $p_2 = \frac{1}{2} + \frac{\epsilon_{p_2}}{2}$, where the correlations ϵ_{p_1} and ϵ_{p_2} belong to $\mathcal{B} = \{x \in \mathbb{R} : -1 \leq x \leq 1\}$. Then, they define the continuous generalization of $\odot$ as $\epsilon_x \odot_{\mathcal{C}} \epsilon_{p_2} = \epsilon_{p_3} = \frac{\epsilon_{p_1}\epsilon_{p_2} + \epsilon_{p_1} + \epsilon_{p_2} - 1}{2}$. By assuming similar independence properties among the input variables, they were able to generalize various cryptographic operations, leading to the creation of continuous versions of entire cryptographic algorithms.

Upon these continuous versions of cryptographic algorithms, they construct three metrics, namely Continuous Avalanche Factor (CAF), Continuous Neutrality Measure (CNM), and Diffusion Factor (DF). The CAF is the continuous equivalent of the avalanche factor [28], which measures the proportion of output bits that change for input Hamming distances equal to 1 on average; this proportion is expected to be 0.5 for a random permutation. In the continuous version, since there is no concept of Hamming distance, the Euclidean Distance (ED) is used to evaluate CAF. The idea behind CAF is to measure how much the output of a continuous version of an algorithm changes, on average, when the input bit's probability of being equal to 1 of a chosen random bit is slightly altered by a small real number λ. In other words, we need to evaluate, on average, the behavior of the ED between the outputs $y_0 = f(x_0)$ and

$y_1 = f(x_1)$ for $x_0, x_1 \in \mathbb{B}$, when the ED of x_0 and x_1 is lesser than λ. It is expected for "good ciphers" that even with small values of λ, higher values on the ED of the propagation of these alterations, on average. For more information on the other two metrics (CNM and DF), see [25].

Within the continuous diffusion test module, CLAASP implements the continuous versions of several cryptographic operations, following Theorem 1 and Definitions 1 to 12 from [25], which can be combined to obtain the continuous version of entire primitives. For instance, the CAF can be evaluated on 8 rounds of Speck128-128, using 10,000 random samples and a Euclidean distance of 0.001 on each input bit, as follows:

```
sage: from claasp.ciphers.block_ciphers.speck_block_cipher import
    ↪ SpeckBlockCipher
sage: speck_cipher = SpeckBlockCipher(number_of_rounds=8, block_bit_size=128,
    ↪  key_bit_size=128)
sage: caf = speck_cipher.continuous_avalanche_factor(0.001, 10000)
sage: caf['plaintext']['cipher_output']['continuous_avalanche_factor']['
    ↪ values'][0]['value']
0.067
```

`caf['plaintext']['cipher_output']['continuous_avalanche_factor']` is an array of dictionaries, where index 0 contains the CAF for Speck128-128 reduced to 1 round, index 1 contains the CAF for Speck128/128 reduced to 2 rounds, and so on. In this case, we obtained a value of 0.067 for CAF at round 8.

The performance of Speck128-128, AES-128, the iterated permutations in ASCON320 and the iterated permutation in ChaCha with respect to CAF, subject to $\lambda = 0.001$, is presented in Table 5. For the iterated permutation in ChaCha, a single round is equivalent to four half-quarter rounds in the table. Figure 8 displays the timing comparison of these ciphers for various sample sizes used in computing CAF. The experiments were conducted on a Ubuntu 22.04.1 machine equipped with 256 AMD core processors and 1TB of memory.

When comparing Table 5 to Table 2 in [25], we observed slight variations in the CAF values reported in Figure 8 compared to the values presented in [25]. This difference is due to our use of the Python Decimal package to handle small numbers, while the implementation of Table 2 in [25] employed the Relic library [3]. For instance, for five rounds of AES-128, we obtained a value of 0.777, whereas [25] reports 0.734.

Rounds	AES	ASCON	ChaCha	Speck
1 to 4	0	0	0	0
5	0.777	0.008	0	0
6	0.971	0.761	0.019	0
7	0.999	0.962	0.257	0.002
8	-	0.998	0.694	0.067
9	-	0.999	0.939	0.318
10	-	-	0.993	0.613
11	-	-	-	0.828
12	-	-	-	0.941
13	-	-	-	0.98
14	-	-	-	0.997

Table 5: Continuous Avalanche Factor comparison for AES-128, ASCON320 permutation, ChaCha permutation, and Speck128-128 using $\lambda = 0.001$.

4.4 Constraint solvers

In the previous section, we mention differential propagation through avalanche properties. While these properties give information on the diffusion of the cryptographic primitive, cryptographers are also interested in properties that cover more rounds, but with lower probability, such as differential or linear characteristics. Finding such characteristic is a difficult combinatorial problem, traditionally handled with Matsui's algorithm [46] variations. In recent years, Matsui's algorithm has been less widely used, in favor of automatic search paradigms, such as Mixed Integer Linear Programming (MILP), SAT, SMT, and more recently Constraint Programming (CP). These tools have the benefit of being extensively studied and optimized by the AI and OR communities, so that the focus shifts from implementing a search algorithm to modeling the problem properly. CLAASP can automatically generate MILP, SAT, SMT and CP models for differential and linear cryptanalysis, from a primitive's description.

The models generated by CLAASP follow state-of-the-art techniques for each of the components, which we cannot enumerate here for space reasons, but will be detailed in a separate work. Our goal is to include further techniques, following the state of the art as new approaches are published, to enable easy comparison.

In practice, the library currently exposes functions that generate models in either paradigm to:

- find one optimal differential or linear trail;
- enumerate all differential or linear trail with a fixed objective value;
- enumerate all differential or linear trails with an objective value better than a given bound.

These functions allow to easily search for optimal trails. In addition, CLAASP implements

- `fix_variables_value_xor_differential_constraints` and
- `fix_variables_value_xor_linear_constraints`,

which permits to specify fixed values for some of the variables. These, combined with the enumeration of trails with conditions on the objective value, permit the evaluation of the probability of differentials, as in [2] for instance. Furthermore, fixing the input and output difference, or linear masks, permits to exhibit impossible differentials or zero-correlation linear approximations, in a similar fashion to [26].

The provided script, `find_differential_related_trails.sage`, demonstrates these functionalities for differential properties, while the corresponding functions for linear properties adhere to the same pattern.

4.5 Algebraic module

The objective of this module is to study the algebraic properties of a specified cipher and test if it is secure against algebraic attacks. In algebraic cryptanalysis, breaking a block or stream cipher, essentially involves solving a set of multivariate polynomial equations over a finite field $\mathbb{F}_q$, which often has one or a few solutions in $\mathbb{F}_q$. But solving a system of multivariate random polynomials is generally a hard task.

This module generates a multivariate algebraic polynomial system corresponding to the "sbox", "linear_ layer", "mix_column", and "constant" components, together with the "XOR", "AND", "OR", "SHIFT", "ROTATE", and "NOT" operations. It provides a

set of polynomials representing the components and operations involved in a particular input cipher along with connection polynomials, which represent the links between the various components. From the polynomial system, it is possible to retrieve its algebraic degree, number of polynomials, and number of variables in order to analyze its algebraic features and the difficulty of solving the system. The security of a cipher (up to a particular number of rounds) against algebraic attacks could be evaluated by solving the corresponding algebraic system up to that many rounds. The module now offers a method to test it by solving the system in a time limit using only the Gröbner basis computation [23] available on the SAGE platform. Consider the following example, for instance:

```
sage: from claasp.ciphers.block_ciphers.present_block_cipher import
    ↪ PresentBlockCipher
sage: from claasp.cipher_modules.models.algebraic.algebraic_model import
    ↪ AlgebraicModel
sage: from claasp.cipher_modules.algebraic_tests import algebraic_tests
sage: algebraic_tests( PresentBlockCipher(number_of_rounds=1), 120)
{'input_parameters': {'timeout': 120},
 'test_results': {'number_of_variables': [1183],
  'number_of_equations': [1328],
  'number_of_monomials': [1660],
  'max_degree_of_equations': [2],
  'test_passed': [False]}}
```

The test results in False, indicating that the system can be solved in the time limit given and that the cipher is not algebraically secure. Note that the test result of True does not guarantee that the encryption is secure for that many rounds. The algebraic module is currently in its preliminary stage and will be improved in upcoming releases.

4.6 Neural aided cryptanalysis module

Following Aron Gohr's seminal paper at CRYPTO'19 [35], improving the state-of-the-art differential cryptanalysis result on the SPECK32-64 cipher, neural-based approaches to cryptanalysis have gained traction in the community. In Gohr's approach, a neural network is trained to distinguish, from an input composed of 2 ciphertexts in binary format, whether they correspond to the encryption of two unrelated plaintexts, or of two plaintexts with a given XOR difference. CLAASP implements such approaches, and other neural-based analysis tools, in `claasp.cipher_modules.neural_network_tests`.

Single ciphertext approach: Neural Network Black box Distinguisher Tests Differential neural cryptanalysis examines pairs of plaintexts. The black box test implemented by CLAASP takes a step back, and focuses on single ciphertexts. Built from [15], this test investigates whether a neural network can find a relation between the inputs of a primitive and its output. The neural network is trained to label samples $[P, C]$ as 0 (if Y is random) or 1 if Y is the output of a given component of the primitive. This test returns a dictionary containing the test parameters, and, for each pair $(in \in [\texttt{plaintext, key}], out)$, the obtained accuracy, for each round output and round key output out. After a certain amount of rounds, the accuracy will converge to 0.5, meaning that the black box distinguisher is not able to distinguish the cipher output from random. The corresponding library function is `neural_network_blackbox_distinguisher_tests`.

Pairs of Ciphertexts: Neural Network Differential Distinguisher Tests This test implements the neural distinguisher described by Gohr in [35], with the simplified training pipeline described in [14], where a depth-1 neural distinguisher trained on n rounds is iteratively retrained for $n+1, \ldots n+t$ rounds, where $n+t$ is the first round where the neural distinguisher fails to learn. Specifically, the neural distinguisher is trained to label samples $[C_0 = E_K(P_0), C_1 = E_K(P_1)]$ as 0 (if $P_0 \oplus P_1$ is random) or 1 if $P_0 \oplus P_1$ is a given, fixed value δ. The corresponding library function is `neural_staged_training`.

Helper Function: Truncated Differential Search For Neural Distinguishers The previous test relies on an input difference with good propagation properties. It has been observed [35] that the input difference that starts the most likely differential does not result in the best neural distinguishers. Further research [16] suggested differential-linear properties, based on highly likely truncated differentials a few rounds before the studied round, may be at play. This assumption was used as the basis to an input difference search technique [14], where a genetic algorithm explores potential input differences and ranks them based on the cumulative biases of the resulting output difference bits. This algorithm is implemented by CLAASP, and can be used to retrieve Gohr's original input difference. It is implemented in the library function `find_good_input_difference_for_neural_distinguisher`.

These functions are illustrated in the `neural_network_based_tests.sage` script of the supplementary material. This script first runs the black box test on 1 round of Speck64, then runs the input difference search for Speck64, and trains Gohr's neural network using the optimal difference returned by the optimizer. Note that the optimizer is not deterministic, and its parameters are adapted for a reasonably fast execution time for demonstration purposes; therefore, it may, in some rare instance, fail to find the optimal input difference $0x00400000$.

5 Benchmark comparison with other libraries

In this section we provide a comparison with libraries that aims at achieving similar goals as CLAASP.

5.1 TAGADA

The TAGADA library focuses on the differential cryptanalysis of word-oriented ciphers with an SPN structure. For such ciphers, it is common (e.g, [20]) to divide the search into two steps. The first step aims to find truncated differential characteristics through the minimization of the non-linear operators utilized in this process. The second step enumerates the truncated differential characteristic passing to the minimum number of non-linear operators found in the previous step. It was shown [34] that the filtering of the first step may be insufficient so that too many solutions are left to explore in step 2. More advanced filtering is, therefore, beneficial and enables scaling to more rounds. This is done through additional constraints that capture linear dependencies between variables during step 1. The TAGADA library generalizes such constraints, making it very efficient for word-based ciphers. These techniques are not, at the moment, included in CLAASP, so TAGADA is expected to perform significantly better on word-based

characteristics search. We are planning to include these additional constraints in the next releases of CLAASP.

On the other hand, the basic version of the first step, searching for the minimum number of active SBoxes of SPN ciphers, is implemented in CLAASP as

`CpXorDifferentialTrailSearchFixingNumberOfActiveSboxesModel`.

TAGADA implements the option of running the first step search with the basic technique used in CLAASP; we attempted to run the search for 3 and 4 rounds of AES-128, but we were not able to reproduce the known results from [41,48,33] with TAGADA, which reported 2 and 7 SBoxes respectively, rather than the expected 3 and 9. On the other hand, CLAASP returned the expected solution. Note that TAGADA can only generate MiniZinc models, while CLAASP allows to directly write the model in the language supported by the solvers (including a MiniZinc interface).

5.2 CASCADA

We make a comparison between CLAASP and CASCADA by taking the time they spend searching for optimal characteristics in the single-key scenario and in the following ciphers: Speck32-64, Speck64-128 and LEA. Specifically, in Figure 9, we show the time spent by CASCADA and CLAASP in the search for an optimal characteristic on across several rounds and using the following SMT solvers: MathSAT, Yices, and Z3. In order to get timings for every round we take the average amount of five repetitions. The experiments were conducted on a machine running Ubuntu 22.04.1, equipped with 256 AMD core processors and 1TB of memory. As observed, while using the Yices solver, the CLAASP library performs similarly to CASCADA. Nevertheless, for MathSAT and Z3, CLAASP exhibits better performance.

In terms of functionalities, CASCADA includes the search for impossible differentials, in particular through the method of [26]. In this method, the variables corresponding to the input and output differences of a differential are fixed to a value that the analyst wants to test, and the solver is run. If the solver finds a solution, then the differential is possible; otherwise, it is impossible. In this method, the analyst usually tests all the pairs of input and output differences of low hamming weight (typically 1). A similar technique can be used for zero-correlation linear approximations. Using this method, CLAASP can for instance retrieve the 17-rounds impossible differential on HIGHT presented in [26] in under 10 minutes on a single core.

6 Conclusion

The fast-paced publication of new cryptanalysis techniques, of improvement of existing ones, makes it crucial to have an efficient way to test a given property on a large number of primitives; CLAASP aims to fulfill this need. In its current form, it already offers a vast array of cipher analysis techniques, from component analysis, to automatic models building, through neural cryptanalysis. Future releases will add more primitives, as well as further analysis techniques, such as guess-and-determine or meet-in-the-middle techniques. More importantly, the CLAASP team is strongly committed to include new state-of-the-art improvements to automated techniques as it evolves, and provide a one-stop shop to evaluate, compare and experiment with modifications on existing methods. Finally, the open-source status of the library is an invitation to researchers from the community to not only use, but also improve CLAASP as they see fit.

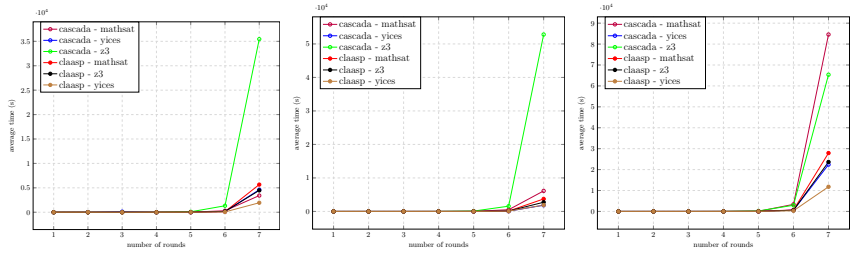


Fig. 9: Time comparison CLAASP vs CASCADA to search for optimal differential characteristics on Speck32-64 (left), Speck64-128 (middle) and LEA128-128(right), using different SMT solvers.

Ackowledgments

This initial version of the CLAASP project was entirely funded by the Technology Innovation Institute of Abu Dhabi (TII) , and it involved several collaborators across a time span of about 4 years. In particular, we would like to thank researchers and professors from the University of Milan, the Politecnico di Torino, the Radboud University, the developers of LeanMind and the DevOps team from TII. The many professors, researchers, and developers who contributed to the project include: Rusydi Makarim, Paul Huynh, Anna Hambitzer, Alessandro De Piccoli, Sergio Polese, Mattia Formenti, Simone Pellizzola, Yousef Hammar, Luca Torresetti, Matteo Rossi, Matteo Protopapa, Maria Dura, Maria Guerra, Yessica Ramos, Ana Cáceres, Joan Daemen, Thomas Peyrin.

References

1. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 430–454. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
2. Ankele, R., Kölbl, S.: Mind the Gap - A Closer Look at the Security of Block Ciphers against Differential Cryptanalysis. In: Cid, C., Jr., M.J.J. (eds.) Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11349, pp. 163–190. Springer (2018). https://doi.org/10.1007/978-3-030-10970-7_8, https://doi.org/10.1007/978-3-030-10970-7_8
3. Aranha, D.F., Gouvêa, C.P.L., Markmann, T., Wahby, R.S., Liao, K.: RELIC is an Efficient Library for Cryptography. https://github.com/relic-toolkit/relic
4. Aumasson, J., Meier, W., Phan, R.C., Henzen, L.: The Hash Function BLAKE. Information Security and Cryptography, Springer (2014). https://doi.org/10.1007/978-3-662-44757-4, https://doi.org/10.1007/978-3-662-44757-4
5. Azimi, S.A., Ranea, A., Salmasizadeh, M., Mohajeri, J., Aref, M.R., Rijmen, V.: A bit-vector differential model for the modular addition by a constant and its applications to differential and impossible-differential cryptanalysis. Des. Codes Cryptogr. **90**(8), 1797–1855 (2022). https://doi.org/10.1007/s10623-022-01074-8, https://doi.org/10.1007/s10623-022-01074-8

6. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A Block Cipher for Low Energy. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology – ASIACRYPT 2015. pp. 411–436. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
7. Banik, S., Pandey, S.K., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y.: GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10529, pp. 321–345. Springer (2017). https://doi.org/10.1007/978-3-319-66787-4_16, https://doi.org/10.1007/978-3-319-66787-4_16
8. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
9. Bassham, L., Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Leigh, S., Levenson, M., Vangel, M., Heckert, N., Banks, D.: Special Publication (NIST SP) - 800-22 Rev 1a: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications (September 2010), https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=906762
10. Bassham, L., Soto, J.: NISTIR 6483: Randomness testing of the advanced encryption standard finalist candidates. NIST Internal or Interagency Reports (2000)
11. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015. pp. 175:1–175:6. ACM (2015). https://doi.org/10.1145/2744769.2747946, https://doi.org/10.1145/2744769.2747946
12. Beierle, C., Biryukov, A., dos Santos, L.C., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q.: Lightweight AEAD and Hashing using the Sparkle Permutation Family. IACR Trans. Symmetric Cryptol. **2020**(S1), 208–261 (2020). https://doi.org/10.13154/tosc.v2020.iS1.208-261, https://doi.org/10.13154/tosc.v2020.iS1.208-261
13. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In: Robshaw, M., Katz, J. (eds.) Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9815, pp. 123–153. Springer (2016). https://doi.org/10.1007/978-3-662-53008-5_5, https://doi.org/10.1007/978-3-662-53008-5_5
14. Bellini, E., Gerault, D., Hambitzer, A., Rossi, M.: A Cipher-Agnostic Neural Training Pipeline with Automated Finding of Good Input Differences. Cryptology ePrint Archive, Paper 2022/1467 (2022), https://eprint.iacr.org/2022/1467, https://eprint.iacr.org/2022/1467
15. Bellini, E., Hambitzer, A., Protopapa, M., Rossi, M.: Limitations Of The Use Of Neural Networks In Black Box Cryptanalysis. In: Innovative Security Solutions for Information Technology and Communications: 14th International Conference, SecITC 2021, Virtual Event, November 25–26, 2021, Revised Selected Papers. p. 100–124. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-031-17510-7_8, https://doi.org/10.1007/978-3-031-17510-7_8
16. Benamira, A., Gérault, D., Peyrin, T., Tan, Q.Q.: A Deeper Look at Machine Learning-Based Cryptanalysis. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on

the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12696, pp. 805–835. Springer (2021). https://doi.org/10.1007/978-3-030-77870-5_28, https://doi.org/10.1007/978-3-030-77870-5_28

17. Bernstein, D.J.: ChaCha, a variant of Salsa20 (2008)
18. Bernstein, D.J., Kölbl, S., Lucks, S., Massolino, P.M.C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F., Todo, Y., Viguier, B.: Gimli : A Cross-Platform Permutation. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10529, pp. 299–320. Springer (2017). https://doi.org/10.1007/978-3-319-66787-4_15, https://doi.org/10.1007/978-3-319-66787-4_15
19. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7881, pp. 313–314. Springer (2013). https://doi.org/10.1007/978-3-642-38348-9_19, https://doi.org/10.1007/978-3-642-38348-9_19
20. Biryukov, A., Nikolic, I.: Automatic Search for Related-Key Differential Characteristics in Byte-Oriented Block Ciphers: Application to AES, Camellia, Khazad and Others. In: Gilbert, H. (ed.) Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6110, pp. 322–344. Springer (2010). https://doi.org/10.1007/978-3-642-13190-5_17, https://doi.org/10.1007/978-3-642-13190-5_17
21. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007. pp. 450–466. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
22. Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: spongent: A Lightweight Hash Function. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6917, pp. 312–325. Springer (2011). https://doi.org/10.1007/978-3-642-23951-9_21, https://doi.org/10.1007/978-3-642-23951-9_21
23. Brickenstein, M., Dreyer, A.: Polybori: A framework for Gröbner-basis computations with Boolean polynomials. J. Symb. Comput. **44**(9), 1326–1345 (2009). https://doi.org/10.1016/j.jsc.2008.02.017, https://doi.org/10.1016/j.jsc.2008.02.017
24. Brown, R.G.: Dieharder: A Random Number Test Suite Version 3.31.1 (2021), available at https://webhome.phy.duke.edu/~rgb/General/dieharder.php
25. Coutinho, M., de Sousa Júnior, R.T., Borges, F.: Continuous Diffusion Analysis. IEEE Access **8**, 123735–123745 (2020). https://doi.org/10.1109/ACCESS.2020.3005504, https://doi.org/10.1109/ACCESS.2020.3005504
26. Cui, T., Chen, S., Fu, K., Wang, M., Jia, K.: New automatic tool for finding impossible differentials and zero-correlation linear approximations. Sci. China Inf. Sci. **64**(2) (2021). https://doi.org/10.1007/s11432-018-1506-4, https://doi.org/10.1007/s11432-018-1506-4

27. Daemen, J., Hoffert, S., Assche, G.V., Keer, R.V.: The design of Xoodoo and Xoofff. IACR Trans. Symmetric Cryptol. **2018**(4), 1–38 (2018). https://doi.org/10.13154/tosc.v2018.i4.1-38, https://doi.org/10.13154/tosc.v2018.i4.1-38
28. Daum, M.: Cryptanalysis of Hash functions of the MD4-family (2005)
29. Dinu, D., Perrin, L., Udovenko, A., Velichkov, V., Großschädl, J., Biryukov, A.: Design strategies for ARX with provable bounds: Sparx and LAX. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10031, pp. 484–513 (2016). https://doi.org/10.1007/978-3-662-53887-6_18, https://doi.org/10.1007/978-3-662-53887-6_18
30. Dobraunig, C., Eichlseder, M., Mendel, F.: Heuristic Tool for Linear Cryptanalysis with Applications to CAESAR Candidates. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9453, pp. 490–509. Springer (2015). https://doi.org/10.1007/978-3-662-48800-3_20, https://doi.org/10.1007/978-3-662-48800-3_20
31. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2: Lightweight authenticated encryption and hashing. J. Cryptol. **34**(3), 33 (2021). https://doi.org/10.1007/s00145-021-09398-9, https://doi.org/10.1007/s00145-021-09398-9
32. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The skein hask function family (2009)
33. Fouque, P., Jean, J., Peyrin, T.: Structural Evaluation of AES and Chosen-Key Distinguisher of 9-Round AES-128. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 183–203. Springer (2013). https://doi.org/10.1007/978-3-642-40041-4_11, https://doi.org/10.1007/978-3-642-40041-4_11
34. Gérault, D., Lafourcade, P., Minier, M., Solnon, C.: Computing AES related-key differential characteristics with constraint programming. Artif. Intell. **278** (2020). https://doi.org/10.1016/j.artint.2019.103183, https://doi.org/10.1016/j.artint.2019.103183
35. Gohr, A.: Improving Attacks on Round-Reduced Speck32/64 Using Deep Learning. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11693, pp. 150–179. Springer (2019). https://doi.org/10.1007/978-3-030-26951-7_6, https://doi.org/10.1007/978-3-030-26951-7_6
36. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In: Rogaway, P. (ed.) Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6841, pp. 222–239. Springer (2011). https://doi.org/10.1007/978-3-642-22792-9_13, https://doi.org/10.1007/978-3-642-22792-9_13
37. Hadipour, H., Eichlseder, M.: Autoguess: A Tool for Finding Guess-and-Determine Attacks and Key Bridges. In: Ateniese, G., Venturi, D. (eds.) Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13269, pp.

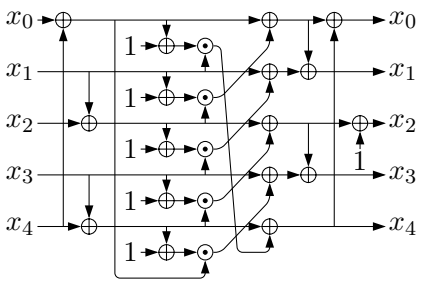
230–250. Springer (2022). https://doi.org/10.1007/978-3-031-09234-3_12, https://doi.org/10.1007/978-3-031-09234-3_12
38. Hell, M., Johansson, T., Maximov, A., Meier, W., Sönnerup, J., Yoshida, H.: Grain-128AEADv2 - A lightweight AEAD stream cipher (2019), https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf
39. Hong, D., Lee, J.K., Kim, D.C., Kwon, D., Ryu, K.H., Lee, D.G.: Lea: A 128-bit block cipher for fast encryption on common processors. In: Kim, Y., Lee, H., Perrig, A. (eds.) Information Security Applications. pp. 3–27. Springer International Publishing, Cham (2014)
40. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: A new block cipher suitable for low-resource device. In: Goubin, L., Matsui, M. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4249, pp. 46–59. Springer (2006). https://doi.org/10.1007/11894063_4, https://doi.org/10.1007/11894063_4
41. Khoo, K., Lee, E., Peyrin, T., Sim, S.M.: Human-readable Proof of the Related-Key Security of AES-128. IACR Trans. Symmetric Cryptol. **2017**(2), 59–83 (2017). https://doi.org/10.13154/tosc.v2017.i2.59-83, https://doi.org/10.13154/tosc.v2017.i2.59-83
42. Leurent, G.: Analysis of Differential Attacks in ARX Constructions. In: Wang, X., Sako, K. (eds.) Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7658, pp. 226–243. Springer (2012). https://doi.org/10.1007/978-3-642-34961-4_15, https://doi.org/10.1007/978-3-642-34961-4_15
43. Leurent, G.: Construction of Differential Characteristics in ARX Designs Application to Skein. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 241–258. Springer (2013). https://doi.org/10.1007/978-3-642-40041-4_14, https://doi.org/10.1007/978-3-642-40041-4_14
44. Libralesso, L., Delobel, F., Lafourcade, P., Solnon, C.: Automatic Generation of Declarative Models For Differential Cryptanalysis. In: Michel, L.D. (ed.) 27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021. LIPIcs, vol. 210, pp. 40:1–40:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021), https://doi.org/10.4230/LIPIcs.CP.2021.40
45. Marsaglia, G.: The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness (1995), web archived at https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/
46. Matsui, M.: Linear Cryptanalysis Method for DES Cipher. In: Helleseth, T. (ed.) Advances in Cryptology — EUROCRYPT '93. pp. 386–397. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)
47. Mouha, N., Preneel, B.: A Proof that the ARX Cipher Salsa20 is Secure against Differential Cryptanalysis. IACR Cryptol. ePrint Arch. p. 328 (2013), http://eprint.iacr.org/2013/328
48. Mouha, N., Wang, Q., Gu, D., Preneel, B.: Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming. In: Wu, C., Yung, M., Lin, D. (eds.)

Information Security and Cryptology - 7th International Conference, Inscrypt 2011, Beijing, China, November 30 - December 3, 2011. Revised Selected Papers. Lecture Notes in Computer Science, vol. 7537, pp. 57–76. Springer (2011). https://doi.org/10.1007/978-3-642-34704-7_5, https://doi.org/10.1007/978-3-642-34704-7_5
49. National Institute of Standards and Technology: Data encryption standard (des). FIPS Publication 46-3 (October 1999)
50. Nethercote, N., Stuckey, P.J., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming - CP 2007. Principles and Practice of Constraint Programming 2007, Springer (2007). https://doi.org/10.1007/978-3-540-74970-7_38, https://doi.org/10.1007/978-3-540-74970-7_38
51. Polimón, J., Castro, J.C.H., Estévez-Tapiador, J.M., Ribagorda, A.: Automated design of a lightweight block cipher with Genetic Programming. Int. J. Knowl. Based Intell. Eng. Syst. **12**(1), 3–14 (2008), http://content.iospress.com/articles/international-journal-of-knowledge-based-and-intelligent-engineering-systems/kes00141
52. Ranea, A., Liu, Y., Ashur, T.: An Easy-to-Use Tool for Rotational-XOR Cryptanalysis of ARX Block Ciphers. IACR Cryptol. ePrint Arch. p. 727 (2020), https://eprint.iacr.org/2020/727
53. Ranea, A., Rijmen, V.: Characteristic automated search of cryptographic algorithms for distinguishing attacks (CASCADA). IET Inf. Secur. **16**(6), 470–481 (2022). https://doi.org/https://doi.org/10.1049/ise2.12077
54. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, N., Dray, J., Vo, S.: Special Publication (NIST SP) - 800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications (May 2001)
55. Sasaki, Y., Todo, Y.: New Impossible Differential Search Tool from Design and Cryptanalysis Aspects - Revealing Structural Properties of Several Ciphers. In: Coron, J., Nielsen, J.B. (eds.) Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III. Lecture Notes in Computer Science, vol. 10212, pp. 185–215 (2017). https://doi.org/10.1007/978-3-319-56617-7_7, https://doi.org/10.1007/978-3-319-56617-7_7
56. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N.: The Twofish Encryption Algorithm: A 128-Bit Block Cipher. John Wiley & Sons, Inc., USA (1999)
57. Soto, J.: NISTIR 6390: Randomness testing of the advanced encryption standard candidate algorithms. NIST Internal or Interagency Reports (1999)
58. Soto, J.: Statistical testing of random number generators. In: Proceedings of the 22nd national information systems security conference. vol. 10, p. 12. NIST Gaithersburg, MD (1999), https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1999/10/21/proceedings-of-the-22nd-nissc-1999/documents/papers/p24.pdf
59. of Standards, N.I., Technology: Advanced encryption standard. NIST FIPS PUB 197 (2001)
60. Stankovski, P.: Automated algebraic cryptanalysis. pp. 11–11. ECRYPT II (2010), tools for Cryptanalysis 2010 ; Conference date: 22-06-2010 Through 23-06-2010
61. Stankovski, P.: Greedy Distinguishers and Nonrandomness Detectors. In: Gong, G., Gupta, K.C. (eds.) Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15,

2010. Proceedings. Lecture Notes in Computer Science, vol. 6498, pp. 210–226. Springer (2010). https://doi.org/10.1007/978-3-642-17401-8_16, https://doi.org/10.1007/978-3-642-17401-8_16
62. Stefan Kölbl: CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives, https://github.com/kste/cryptosmt
63. Vesselinux, Laboratory of Algorithmics, C., of Luxembourg University, S.L.: Vesselinux/yaarx: Yet another toolkit for analysis of ARX cryptographic algorithms, https://github.com/vesselinux/yaarx
64. Wheeler, D.J., Needham, R.M.: Tea, a tiny encryption algorithm. In: Preneel, B. (ed.) Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings. Lecture Notes in Computer Science, vol. 1008, pp. 363–366. Springer (1994). https://doi.org/10.1007/3-540-60590-8_29, https://doi.org/10.1007/3-540-60590-8_29
65. Wheeler, D.J., Needham, R.M.: Tea extensions (1997)
66. Wu, H., Huang, T.: TinyJAMBU : A Family of Lightweight Authenticated Encryption Algorithms (Version 2) (2019)

A Two cipher representations of ASCON

The nonlinear layer of ASCON permutation can be represented as circuit made of word operation components (XOR, AND and NOT) or with a layer of parallel S-boxes. This is detailed in Figure 10.




```
ASCON Sbox = [
    0x04, 0x0b, 0x1f, 0x14,
    0x1a, 0x15, 0x09, 0x02,
    0x1b, 0x05, 0x08, 0x12,
    0x1d, 0x03, 0x06, 0x1c,
    0x1e, 0x13, 0x07, 0x0e,
    0x00, 0x0d, 0x11, 0x18,
    0x10, 0x0c, 0x01, 0x19,
    0x16, 0x0a, 0x0f, 0x17
]
```

Fig. 10: Two equivalent cipher representation in ASCON. The left figure is the circuit that represents ASCON S-Box (as stated in [31]). The circuit can be seen as NOT, AND and XOR components acting on 64-bit words. The right side is ASCON 5-bit S-Box as an integer list. The nonlinear layer can be seen as the application of 64 parallel S-Boxes. Both cipher representations are implemented in CLAASP.

B Avalanche dataset generation

Given primitive enc, n-bits plaintext P, key $K = 0$, the mask $mask_i$ with 1 at i-bit and others 0, then the avalanche dataset is the concatenation of $enc_K(P) \oplus enc_K(P \oplus mask_i)$ with different P as shown in Figure 11.

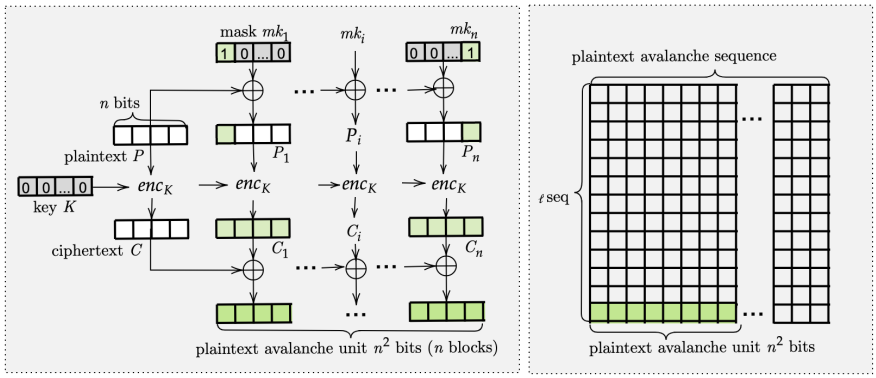


Fig. 11: Illustration of avalanche dataset generation.

C AES as radar charts

D Heatmap of avalanche entropy vectors

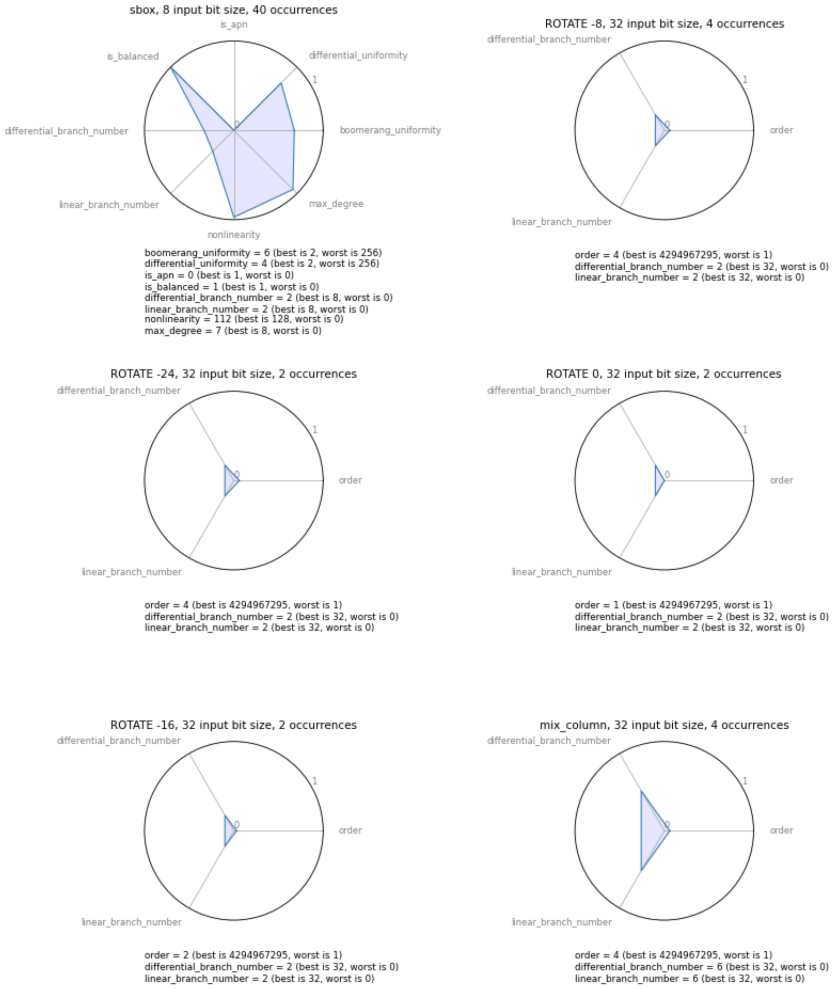


Fig. 12: AES main components as radar charts. The outer region of the radar represents the best value for any property.

Fig. 13: ASCON320 - avalanche entropy vectors - difference injected in position 0 of plaintext with 10000 samples