The \texttt{booleantools} Package: An Open-Source Python Framework for Boolean Functions

Andrew Penland and Wesley Rogers

\textbf{Abstract.} Boolean functions are crucial in the design of secure cryptographic algorithms. We introduce \texttt{booleantools}, an open-source Python package for the analysis and design of boolean functions. As an example of the software’s functionality, we show how it can be used to find geometric information about the space of all boolean functions on 5 variables.

1. Introduction

\textit{Boolean functions} are one of the mathematical building blocks used in the construction of algorithms for cryptography and coding theory. As just one example, boolean functions are used in the \textit{Secure Hash Standard} developed by the National Institute of Standards and Technology and required under federal law for securing classified data (National Institute of Standards and Technology, 2015). In this paper, we present \texttt{booleantools}, a Python package designed to facilitate the computational analysis of boolean functions.

We let $\mathbb{F}_2 = \{0, 1\}$ denote the finite field with two elements, equipped with the usual operations: addition modulo 2, denoted by $\oplus$, and multiplication modulo 2, denoted by juxtaposition. We write $\mathbb{F}_2^n$ for the $n$-dimensional vector space over $\mathbb{F}_2$. A boolean function is a map from $\mathbb{F}_2^n$ to $\mathbb{F}_2$. It is well-known that such a function can always be represented as a polynomial in $n$ variables with coefficients in $\mathbb{F}_2$, and that is the representation we will typically use here. We will provide further mathematical background in Section 3.

There are several properties that are accepted as being necessary for cryptographic security of boolean functions, including \textit{resiliency}, \textit{nonlinearity}, lack of \textit{linear structures}, high \textit{algebraic degree}, etc. We give definitions of these properties in Section 3. We highly recommend Carlet (2010) for an in-depth overview of these topics.

The designer of cryptographically secure boolean functions faces many challenges. Brute force enumeration of all possible functions on $n$ variables becomes infeasible very quickly as $n$ grows. Another challenge is that the various necessary properties are in conflict. A result due to Siegenthaler (1984) shows that as the resiliency of an $n$-variable boolean function increases, the algebraic degree necessarily decreases. Hence, finding suitable boolean functions is a multi-objective optimization problem which requires sophisticated computation in conjunction with mathematical analysis.

The \texttt{booleantools} package offers useful functionality that is not directly available in any existing open source Python package. The Python packages \texttt{PyCrypto} (Litzenberger, 2018) and \texttt{pyca/cryptography} (pyca/cryptography Developers, 2018) offer high-level implementations of cryptographic algorithms for software developers, but they do not make boolean
functions or their properties easily accessible. Another package, PolyBoRi, also works with polynomial functions over \( \mathbb{F}_2 \). The booleanantools package differs from PolyBoRi in two ways: booleanantools is written entirely in Python, and booleanantools has built-in support for many algebraic and computational manipulations of interest in cryptography and coding theory. While packages in GAP (GAP, 2018), SageMath (The Sage Developers, 2018), and other computer algebra systems could be used to do the same things that booleanantools does, they are large multipurpose tools, with large code bases. Since booleanantools is implemented in Python, it can be seamlessly integrated with other Python packages for machine learning (such as Scikit-learn (Pedregosa et al., 2011)) and evolutionary algorithms (such as deap (Fortin et al., 2012)), both of which are established techniques for search and analysis of boolean functions(see, for instance, Asthana et al. (2014) and Sadohara (2001)).

2. Technical Description

The booleanantools package is a Python package that allows the user to create boolean functions using many different representations. Once the boolean function is created, booleanantools provides built-in methods to analyze the function for various properties discussed in the literature. Additionally, booleanantools provides support for actions of permutation groups on functions, and geometric tools such as Hamming and Hausdorff distance functions.

2.1. Installation and Requirements

The booleanantools package is available on PyPi at 
https://pypi.python.org/pypi/booleanantools
and can be installed using the standard Python package manager pip, by typing

```
pip3 install booleanantools
```

from a command line prompt.

The booleanantools package requires any version of Python 3. For Python versions before Python 3.3, the user may need to first install pip by following the instructions available via 

The commands in this paper will be from booleanantools version 0.4.2.

3. Background & Examples

In this section we discuss properties of boolean functions that are relevant for cryptographic research. For each property, we give the definition, as well as a code snippet demonstrating how booleanantools can be used to compute this property. Our discussion of these properties follows Chapter 4 of Carlet (2010).

We write \([n]\) for the set \( \{0, 1, \ldots, n - 1\} \). We will use the vector notation \( \mathbf{x} \) to indicate an element of a vector space \( \mathbb{F}_2^n \), with the variable \( x_i \) representing the \( i \)’th coordinate of \( \mathbf{x} \) for \( i \in [n] \). A monomial is the product of the elements in some subset of \( \{x_0, x_1, \ldots, x_{n-1}\} \).

It is well-known that any boolean function from \( \mathbb{F}_2^n \) can be represented as a polynomial on the variables \( x_0, x_1, \ldots, x_{n-1} \) using Lagrange interpolation (see Lidl and Niederreiter (1994)). This leads to writing boolean functions as a sum of monomials, and that is the representation we typically use for booleanantools.
In some instances, it is helpful to identify a vector in $\mathbb{F}_2^n$ with its integer interpretation as a binary number, via the correspondence $x \iff \sum_{k=0}^{n-1} x_k 2^k$. For instance, under this representation, the vector $[1, 0, 1, 0]$ in $\mathbb{F}_2^n$ would correspond to $(1)2^0 + (0)2^1 + (1)2^2 + (0)2^3 = 6$. This facilitates the representation of an $n$-variable boolean functions via a rule table, a list of length $L = 2^n$ with indices taken from 0 to $L-1$, and the value at the $k$'th index given by the value of $f$ on the boolean vector corresponding to $k$. As an example, the 4-variable boolean function $f(x)$ given by $f(x) = x_0x_1$ would have a rule table of:

<table>
<thead>
<tr>
<th>$x$ (integer form)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Before implementing any of the code examples below, the user should import `booleantools` into their Python file or session. One approach would be:

```python
from booleantools import *
```

For the remainder of the paper, we will use the functions $g$ and $h$ as examples, defined as follows.

$x = (x_0, x_1, \ldots, x_4)$

$g(x) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0x_1$

$h(x) = x_0x_1 \oplus x_2x_3 \oplus x_4$

This is easily translated to `booleantools` code.

```python
>>> x = getX(5)
>>> g = BooleanFunction([[1],[2],[3],[4],[0,1]], 5)
```

A BooleanFunction object can be evaluated at a given point by treating the object as a function and passing in either the vector entries directly or a list of values.

Below are two different methods of evaluating the function $g$ at a point.

```python
>>> g(1,1,0,0,0)
>>> g([1,0,1,0,1])
```

0

0

The Hamming weight of an $n$-variable boolean function $f$ is defined as the number of elements in $\mathbb{F}_2^n$ that $f$ maps to 1. We write $\text{wt}(f)$ the Hamming weight of $f$. An $n$-variable boolean function $f$ is balanced if $\text{wt}(f) = 2^{n-1}$, i.e. if the number of elements that $f$ maps to 1 is exactly the half the size of the domain. This is implemented in `booleantools` through the function `is_balanced`.

```python
>>> g.is_balanced()
>>> h.is_balanced()
```
The Hamming distance between two $n$-variable boolean functions $r$ and $q$ is defined as

$$d(r, q) = \text{wt}(r \oplus q)$$

and gives the number of elements of $\mathbb{F}_2^n$ for which $r$ and $q$ disagree. Hamming distance is implemented in booleantools as follows:

```python
>>> g.hamming_distance(h)
16
```

The dot product on $\mathbb{F}_2^n$ is defined as $d(r, q) \pmod{2}$ and denoted by $r \cdot q$.

The Walsh transform is a useful tool in the study of boolean functions. For an $n$-variable boolean function $f$, the Walsh transform of $f$, denoted by $Wf$, is a real-valued function defined for a vector $v \in \mathbb{F}_2^n$ by:

$$(Wf)(v) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \cdot x \cdot v}.$$ 

```python
>>> g.walsh_transform()
[0, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 0, 0, -16, 0, 0, 0, 0, 0, 0, 0, 16]
```

A boolean function $f$ is called $k$-th order correlation immune if when we hold any $k$ variables constant, the result (viewed as a function on $n - k$ variables) has the same proportion of 1’s in the output as the original function. Our implementation for determining $k$-order correlation immunity is based on the Walsh transform, utilizing a criterion established by Xiao and Massey (1988), which says that a function $f$ is $k$-correlation immune if and only if $Wf(v) = 0$ whenever $\text{wt}(v) \leq k$.

```python
>>> g.is_correlation_immune(k=2)
>>> h.is_correlation_immune(k=2)
```

A boolean function $f$ is $k$-resilient if it is balanced and $k$’th-order correlation immune. We note that $g(x)$ is 2-resilient, and $h(x)$ is only 1-resilient.

```python
>>> g.is_k_resilient(k=2)
>>> h.is_k_resilient(k=2)
```

If $x_i$ is a variable such that flipping the value of $x_i$ (i.e. replacing $x_i$ with $x_i \oplus 1$) also changes the value of $f(x)$ for all $x \in \mathbb{F}_2^n$, then $x_i$ is called a linear structure for the function $f$.

The method linear_structures returns the linear structures of a boolean function, as a set.

```python
g.linear_structures()
```

{2, 3, 4}
The algebraic degree of a monomial is defined as the number of variables in the product; the degree of $x_1$ is 1, the degree of $x_0x_1x_2$ is 3, etc. The algebraic degree of an arbitrary boolean function is defined as the maximum of the algebraic degrees of its monomials; the algebraic degree of $x_1 \oplus x_2$ is 1, the algebraic degree of $x_1x_2 \oplus x_3$ is 2, etc.

A boolean function is affine if its algebraic degree is equal to 1. The nonlinearity of a boolean function $f$ is defined as the minimum distance from $f$ to any element in the space of affine functions. If $f$ is linear, its nonlinearity is 0. We can use the `booleantools` functions `is_affine` and `nonlinearity` to determine these properties. Our implementation of the `nonlinearity` function also relies on the Walsh transform.

```python
>>> g.nonlinearity()
8
>>> h.nonlinearity()
12
>>> g.is_affine()
False
>>> h.is_affine()
False
```

The Hamming distance leads us to further geometric considerations on the space of $n$-variable boolean functions. The development of the `booleantools` package arose from our interest in studying the space of boolean functions using ideas from geometry and group theory. We now briefly review a few necessary ideas in this vein.

If $X$ is a set, a permutation on $X$ is a bijective function from the set to itself. We write $\text{Sym}(n)$ for the set of all permutations on $[n]$, known as the symmetric group on $[n]$. Let $f$ be a boolean function, $\sigma$ be a permutation in $\text{Sym}(n)$, and $x_i$ an indeterminate in the polynomial ring of $\mathbb{F}_2$. We define the function $f^\sigma$ to be given by $f^\sigma(x_i) = f(x_{\sigma(i)})$, for $i \in [n]$ i.e. the coordinates are permuted before the boolean function is applied.

```python
>>> perms = Sym(5)
>>> h.apply_permutation(perms[0])
BooleanFunction([[0, 1], [2, 3], [4]], 5)
```

The diameter of a set with respect to some distance function is the maximum distance obtained by any pair of points in the set. We can define a distance between two sets of boolean functions as well. If $X$ and $Y$ are two sets of $n$-variable boolean functions, the Hausdorff distance between $X$ and $Y$ is defined as:

$$d_H(X, Y) = \max\{\max_{x \in X} \min_{y \in Y} d(x, y), \max_{y \in Y} \min_{x \in X} d(x, y)\}.$$

Below we show how to obtain the Hausdorff distance between the equivalence classes of our example boolean functions:

```python
>>> hausdorff_distance(g.get_orbit(), h.get_orbit())
12
```
It is worth noting that the nonlinearity of an $n$-variable boolean function $f$ is exactly the Hausdorff distance from the singleton set $\{f\}$ to the set of all affine functions on $n$ variables.

The deep relationships between algebraic structure, geometry, and group theory in coding theory and cryptography remain a subject of active research. We suggest the classic Conway and Sloane (2013) as a starting point for the interested reader.

3.1. Other Functions and Features

In addition to the functions given in the previous subsection, there are a large number of other functions which ease the ability to analyze particular classes of functions. We have included documentation for a list of them below. The most recent documentation is available on the package github page.

**Sym(n)**

**Input:** an integer (int) $n$

**Returns:** a list of all possible permutations as a list

```plaintext
>>> Sym(2)

[(0, 1), (1, 0)]
```

**getX(n)**

**Input:** an integer (int) $n$

**Output:** a list of functions of the form $f(x) = x_i$, for $0 \leq i < n$.

```plaintext
>>> getX(2)

[BooleanFunction([[0]], 1), BooleanFunction([[1]], 2)]
```

**generate_function(rule_number, n)**

**Input:** a rule number, which is an integer given by the base-2 encoding of the rule table.

**Output:** a boolean function on $n$ variables with the specified rule number.

```plaintext
>>> generate_function(120, 3)

BooleanFunction([[1, 2], [0]], 3)
```

**weight_k_vectors(k, nbits)**

**Input:** $k$, the desired weight, and $nbits$, the number of bits

**Output:** a list containing all vectors in $F_2^n$ with weight exactly equal to $k$

```plaintext
>>> weight_k_vectors(2, 3)

[[1, 1, 0], [1, 0, 1], [0, 1, 1]]
```
weight_k_or_less_vectors(k, nbits)

**Input:** $k$, the desired weight, and $nbits$, the number of bits

**Output:** a list containing all vectors in $\mathbb{F}_2^n$ with weight less than or equal to $k$

```python
>>> weight_k_or_less_vectors(2, 3)

[[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0], [1, 0, 1], [0, 1, 1]]
```

duplicate_free_list_polynomials(list_of_polys)

**Input:** A list of polynomials.

**Output:** The duplicate_free_list_polynomials function takes a list of polynomials, and returns the list with duplicates removed.

```python
>>> duplicate_free_list_polynomials([BooleanFunction([[1], [1,2]], 3), BooleanFunction([[1,2], [1]], 3)])

[BooleanFunction([[1], [1, 2]], 3)]
```

orbit_polynomial(polynomial, permset)

**Input:** a polynomial (represented as a BooleanFunction object) and optionally a set of permutations

**Output:** the orbit of the polynomial under the permutation set

```python
>>> orbit_polynomial(BooleanFunction([[1], [2]]), Sym(2))

[BooleanFunction([[1], [1, 2]], 3), BooleanFunction([[0], [2]])]
```

orbit_polynomial_list(polynomial_list, permset)

Similar to orbit_polynomial, but for a list of polynomials.

**Input:** list of polynomials (with each polynomial represented as a list of monomial lists) and a set of permutations

**Output:** the orbit of the polynomials in polynomial_list under the set of all permutations in permset.

```python
>>> orbit_polynomial_list([BooleanFunction([[1], [2]]), BooleanFunction([[0], [2]])], Sym(2))

[BooleanFunction([[1], [1, 2]], 3), BooleanFunction([[0], [2]]), BooleanFunction([[0], [2]])]
```
**siegenthaler_combination**(*f1*,*f2*,new **var**)

**Input:** two *n*-variable boolean functions, *f*<sub>1</sub> and *f*<sub>2</sub>, represented as booleanfunction objects

**Output:** A booleanfunction representing what we call the *Siegenthaler combination* of *f*<sub>1</sub> and *f*<sub>2</sub>, both of which are boolean functions on *n* variables. This was introduced in [Siegenthaler (1984)](https://example.com), and is defined as

\[
g(x) = x_n f_1(x) \oplus (1 \oplus x_n) f_2(x).
\]

The Siegenthaler combination has the property that it increases the number of variables from *n* to *n* + 1, while keeping the resiliency the same, and without introducing any additional linear structures.

```plaintext
1 >>> f1 = BooleanFunction([[1]], 2)
2 >>> f2 = BooleanFunction([[0], [0,1]], 2)
3 >>> nv = BooleanFunction([[2]], 3)
4 >>> siegenthaler_combination(f1, f2, nv)

BooleanFunction([[1, 2], [0], [0, 1], [0, 2], [0, 1, 2]], 3)
```

**generate_all_siegenthaler_combinations**(*func_list*,new **var**)

**Input:** a list of booleanfunction objects

**Output:** a list giving all possible Siegenthaler combinations of the functions, without removing duplicates.

```plaintext
1 >>> f1 = BooleanFunction([[1]], 2)
2 >>> f2 = BooleanFunction([[0], [0,1]], 2)
3 >>> f3 = BooleanFunction([[2]], 2)
4 >>> nv = BooleanFunction([[3]], 3)
5 >>> generate_all_siegenthaler_combinations([f1, f2, f3], nv)

[ BooleanFunction([[1], 4]),
  BooleanFunction([[1, 3], [0], [0, 1], [0, 3], [0, 1, 3]], 4),
  BooleanFunction([[1, 3], [2], [2, 3]], 4),
  BooleanFunction([[0, 3], [0, 1, 3], [1], [1, 3]], 4),
  BooleanFunction([[0], [0, 1]], 4),
  BooleanFunction([[0, 3], [0, 1, 3], [2], [2, 3]], 4),
  BooleanFunction([[2, 3], [1], [1, 3]], 4),
  BooleanFunction([[2, 3], [0], [0, 1], [0, 3], [0, 1, 3]], 4),
  BooleanFunction([[2]], 4)]
```

**reduce_to_orbits**(*f_list*, **permset**)

**Input:** a list of functions *f_list* and a set of permutations **permset**

**Output:** a list of representatives from each class, under the action of **permset** on *f_list*

```plaintext
1 >>> reduce_to_orbits([BooleanFunction([[0]], 2), BooleanFunction([[1]], 2)], Sym(2))

[ BooleanFunction([[0]], 2)]
```
In addition to the methods for the analysis of boolean functions, there are several convenience functions.

**Addition and multiplication of functions**

The addition and multiplication of functions is fully supported, using standard Python notation for addition and multiplication. This is seen above, in the Python construction of function $h$.

**BooleanFunction.tex\_str(math\_mode=False)**

**Output:** a \text{T}_{\text{E}}X representation of this function, along with proper math mode support if \text{math\_mode} is set to True.

**Example:**

```python
>>> print(g.tex\_str ( ))
>>> print(h.tex\_str(math\_mode=True ))
```

When rendered in \text{T}_{\text{E}}X, we get the representations

$$g(x) = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0 x_1$$

$$h(x) = x_0 x_1 \oplus x_2 x_3 \oplus x_4$$

### 3.2. Source Code

The full source code of the \text{booleantools} package is available in \text{Appendix B} and on GitHub at \url{https://github.com/MagicalAsh/BooleanFunctions}.

### 4. Example - Geometry of 2-Resilient Boolean Functions

As an example of the utility of the \text{booleantools} package, we demonstrate how we used it explore the geometry of the space of 2-resilient nonlinear boolean functions on 5 variables. We only consider nonlinear functions, since affine functions are known to be cryptographically insecure.

It should be noted that we are not the first to consider these functions. All 5-variable 2-resilient boolean functions were examined in [Braeken et al. (2005)](Braeken2005), which determined cryptographic properties of all boolean functions on six variables or less.

Using the \text{booleantools} package, we were able to determine all nonlinear 2-resilient Boolean functions on five variables by exhaustive search. We then sorted the functions into their orbits under the symmetric group. The following table summarizes our findings for the orbits of the nonlinear 2-resilient boolean functions on five variables. The code we used is available in \text{Appendix A}, and took approximately 20 minutes to run on a personal computer running Debian 4.14 on a twenty core Intel Xeon. Additionally, after verifying the classes of 2-resilient Boolean functions, we calculated the diameter of each class, along with the linear structures of each class.

The code in \text{Appendix A} works by producing every possible quadratic polynomial on five variables, then sorting the resulting polynomials based on their resiliency. After processing all possible
TABLE 4.1. The class representatives, diameter, orbit size, and linear structures of 5-variable 2-resilient boolean functions

<table>
<thead>
<tr>
<th>Orbit Number</th>
<th>Orbit Representative</th>
<th>Linear Structures</th>
<th>Number in Orbit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0 x_1$</td>
<td>$x_2, x_3, x_4$</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_0 x_2$</td>
<td>$x_1, x_3, x_4$</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_0 x_1 \oplus x_0 x_1$</td>
<td>$x_2, x_3, x_4$</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>$x_2 \oplus x_3 \oplus x_0 x_1 \oplus x_0 x_3 \oplus x_1 x_3$</td>
<td>$x_2, x_4$</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_0 x_1 \oplus x_0 x_4 \oplus x_1 x_4$</td>
<td>$x_2, x_4$</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>$x_1 \oplus x_2 \oplus x_3 \oplus x_0 x_1 \oplus x_0 x_2$</td>
<td>$x_3, x_4$</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>$x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_0 x_2 \oplus x_0 x_1$</td>
<td>$x_2, x_3$</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>$x_1 \oplus x_3 \oplus x_4 \oplus x_0 x_1 \oplus x_0 x_2 \oplus x_1 x_3 \oplus x_2 x_3$</td>
<td>$x_4$</td>
<td>60</td>
</tr>
</tbody>
</table>

It may be executed from a command line as

```
python3 generate_classes.py 5
```

5. Conclusion, Future Work, and an Invitation

The reality is that any software package can be extended and improved. We intend to continue to develop booleantools, adding more support for cryptographic tests, methods for properties from coding theory, and support for group theory and group actions. We also intend to further optimize the existing methods for efficiency, as well as providing more support for multithreading and parallelization.

The code will be maintained at the second author’s GitHub repository (at https://github.com/MagicalAsh/BooleanFunctions) and as a package on PyPi (https://pypi.python.org/pypi/booleantools).

We invite questions, suggestions and feature requests from interested parties. Those interested in contributing code or ideas for improvement are welcome to do so through a pull request at the GitHub repository.
Acknowledgments.

We would like to thank the Department of Mathematics and Computer Science at Western Carolina University for use of its computer laboratory, and Geoff Goehle for advising us on use of the lab. We would also like to thank John Asplund for a careful reading and many useful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 1524607.

References


Appendix A. Code for Class Verification

```python
from booleantools import BooleanFunction
import sys
import booleantools as bt
import itertools
import multiprocessing as mp
import json

# linear part + quadratic part

def analyze_polys(poly_gens, que, thread_no, n):
    for polys in poly_gens:
        for poly in polys:
            func = BooleanFunction(poly, n)
            if func.is_k_resilient(k=n−3) and not func.is_affine():
                que.put(poly)

    # Once all of the generators have executed
    que.put(thread_no)


def reduce_classes_dgen(class_list, new_f, n):
    """
    Produces a *MINIMALLY* reduced function list. This is by no means fully
    reduced.
    """
    for f in class_list:
        if new_f in f:
            return None
    class_list.append(get_class(new_f, n))
    return None


def reduce_classes(func_list):
    class_list = []
    for f in func_list:
        in_one = False
        for g in class_list:
            if f in g:
                in_one = True
                if in_one == False:
                    class_list.append(f.get_orbit())
            return class_list


def get_class(f, n):
    perms = bt.Sym(n)
```
```python
return [apply_permutation(f, sigma) for sigma in perms]

def apply_permutation(poly, perm):
    def apply_perm_to_monomial(perm, monomial):
        out = []
        for var in monomial:
            out.append(perm[var])
        return out

    out = [apply_perm_to_monomial(perm, i) for i in poly]
    return out

def powerset(iterable):
    s = list(iterable)
    return list(  
        itertools.chain.from_iterable(  
            itertools.combinations(s, r)  
            for r in range(len(s)+1)  
        )
    )

def func_generator(lin_part, non_lin_parts):
    lin = [[mon] for mon in lin_part]
    for nonlinear in non_lin_parts:
        yield lin + list(nonlinear)

def main():
    n = int(sys.argv[1])

    nonlin = powerset(itertools.combinations(list(range(n)), 2))
    nonlin.remove(())
    lin = powerset(list(range(n)))

    generator_list = [func_generator(linear, nonlin) for linear in lin]
    size = len(generator_list)//16
    chunked = [generator_list[i:i+size] for i in range(0, len(generator_list), size)]

    que = mp.Queue()
    threads = []
    for generators in chunked:
        thre = mp.Process(target=analyze_polys, args=(generators, que, len(threads), n))
        thre.start()
        threads.append(thre)

    deadCnt = 0
    f_out = []
    while deadCnt < len(chunked):
        f = que.get()
        if isinstance(f, list):
            reduce_classes_dgen(f_out, f, n)
        else:  # it’s an int
            deadCnt += 1
            threads[f].join()

    f_out = reduce_classes([BooleanFunction(f[0], n) for f in f_out])
```

---

**The booleantools Package**

41
with open("out/classes.%d.json" % n, "w") as outfile:
    outfile.write(json.dumps({i: f_out[i][0].listform for i in range(len(f_out))}), indent=4))

if __name__ == "__main__":
    if (len(sys.argv) != 2):
        print("USAGE: python3 generate_classes.py <n>")
    else:
        main()

Appendix B. Full Source Code for boolentools Package

import copy as _copy
import boolentools.fields as _fields
from itertools import combinations as _combs
from itertools import permutations as _perms

def Sym(n):
    """
    Creates a set containing all permutations in the symmetric group $S_n$.
    Returns:
    list: A set containing every permutation in $S_n$, in one-line notation.
    """
    return list(_perms([i for i in range(n)]))

GF2 = _fields.PrimeField(2)

class FieldFunction:
    """
    Represents a function over a Finite Field.
    """

def __init__(self, listform, n, field):
    self.listform = listform
    self.n = n
    self.field = field
    self.__reduce()

def __call__(self, *args):
    args = list(args)
    if len(args) == 1 and hasattr(args[0], '__getitem__'):
        args = args[0]
    elif len(args) == 1 and isinstance(args[0], int):
        args = _dec_to_base(args[0], self.n, self.field.order)
    for pos, val in enumerate(args):  # Simplification for inputs
        args[pos] = self.field.get(val)
    value = self.field.get(0)
    for monomial in self.listform:
if monomial not in self.field:
    prod = self.field.get(1)
    for var in monomial:
        if var not in self.field:
            prod *= args[var]
        else:
            prod *= var
    value += prod
else:
    value += monomial

return self.field.value_of(value)

def apply_permutation(self, perm):
    ""
    Applies a permutation to this function.
    \[
    f^\sigma(x)
    \]
    where $\sigma$ is in one line notation.
    ""
    def apply_perm_to_monomial(perm, monomial):
        out = []
        for var in monomial:
            if var in self.field:
                out.append(var)
            else:
                out.append(perm[var])
        return out

        out = [apply_perm_to_monomial(perm, i) for i in self.listform]
    return FieldFunction(out, self.n, self.field)

def __add__(a, b):
    if a.field != b.field:
        raise ValueError("Summands from different fields.")
    if isinstance(b, fields.PrimeField._FieldElement):
        return FieldFunction(a.listform + [[b]], a.n, a.field)
    else:
        return FieldFunction(a.listform + b.listform, max(a.n, b.n), a.field)

def __mul__(a, b):
    if a.field != b.field:
        raise ValueError("Multiplicands are not from the same field.")
if isinstance(b, _fields.PrimeField._FieldElement):
    out = [monomial + [b] for monomial in a.listform]
    return FieldFunction(out, a.n, a.field)
else:
    out = []
    for monomial in a.listform:
        out += [monomial + monomial_b for monomial_b in b.listform]
    return FieldFunction(out, max(a.n, b.n), a.field)

def tex_str(self, math_mode=False):
    ""
    Creates a TeX String from this FieldFunction.
    Args:
        math_mode (bool, optional): Whether to return with surrounding '$'.
    Returns:
        str: A proper TeX String representing this function.
    ""
    out = '' if not math_mode else "$"
    flag = False
    for monomial in self.listform:
        out += '\oplus ' if flag else ''
        for term in monomial:
            out += 'x_{' + str(term) + '}'
    flag = True
    return out if not math_mode else out + "$"

def __str__(self):
    return self.tex_str()

def __repr__(self):
    return "FieldFunction(%s, %s, %s)" % (str(self.listform), str(self.n),
                                          str(self.field))

def __reduce__(self):
    for i in range(len(self.listform)):
        self.listform[i] = [val for val in self.listform[i] if val != self.field.get(1)]

class BooleanFunction(FieldFunction):
    ""
    This class represents a boolean function ($\mathbb{F}_2^n \rightarrow\mathbb{F}_2$) and implements a large amount of useful functions.
    ""
    def __init__(self, listform, n):
        ""
        Creates a boolean function on n variables.
Attributes:

- `listform (list)`: A list of the monomials this polynomial contains.
  Ex. \([x_1 \oplus x_2 x_3]\) is \([0, 1, 2]\).
- `n (int)`: The number of variables, where \(n - 1\) is the highest term in the list form.

```python
super().__init__(listform, n, GF2)
_copyList = []

# This is done for space efficiency. Basically reduces coefficient mod 2

for i in listform:
    if i not in _copyList:
        _copyList.append(i)
    else:
        _copyList.remove(i)

self.listform = _copyList
self.update_rule_table()
```

```python
def hamming_weight(self):
    """
    Returns the Hamming Weight of this function.
    """
    Returns:
    int: The hamming weight of this function.
    """
    return sum(self.tableform)

def hamming_distance(self, other):
    """
    Determines the hamming distance of a function or a list of functions.
    """
    Args:
    other (BooleanFunction): function or list of functions to find distance to.
    Returns:
    int: A list of distances if #other is a list, or a float if #other is another function.
    """
    if hasattr(other, "__getitem__"):
        if other is a list
            return [self.hamming_distance(f) for f in other]
    else:
        u = self.tableform
        v = other.tableform
        s = sum([_delta(u[k], v[k]) for k in range(len(u))])
        return s

def walsh_transform(self):
    """
    Performs a Walsh transform on this function.
    """
    Returns:
list: A list containing the walsh transform of this function.

```python
f = self.tableform
nbits = self.n
vecs = [(_dec_to_bin(x, nbits), x) for x in range(len(f))]
def Sf(w):
    return sum([-1]**(f[x] * _dot_product(vec, w)) for vec, x in vecs)
Sf_list = [Sf(vec) for vec, x in vecs]
return Sf_list
```

def walsh_spectrum(self):
    """
    Generates the Walsh spectrum for this function.
    Returns:
    float: The Walsh spectrum of this function.
    """
    f = self.tableform
    walsh_transform_f = self.walsh_transform()
    spec = max([abs(v) for v in walsh_transform_f])
    return spec

def is_balanced(self):
    """
    Determines whether this function is balanced or not.
    # Returns
    bool: True if balanced, False otherwise.
    """
    f = self.tableform
    return sum(f) == len(f)/2

def is_correlation_immune(self, k=1):
    """
    Determines if this function is k correlation immune.
    Args:
    k (int): immunity level
    """
    if k > self.n:
        raise BaseException("Correlation immunity level cannot be higher
        than the number of variables.")
    f = self.tableform
    walsh_transform = self.walsh_transform()
    nbits = self.n
    vectors_to_test = [bin_to_dec(vec) for vec in
    weight_k_or_less_vectors(k, nbits)]
    walsh_transform_at_weight_k = [walsh_transform[vec] for vec in
    vectors_to_test]
    return walsh_transform_at_weight_k == [0]*len(walsh_transform_at_weight_k)

def is_k_resilient(self, k=1):
    """
    Determines if this boolean function is k-resilient.
    """
```
Args:
    `k (int)`: immunity level

.. code:: python

    return self.is_balanced() and self.is_correlation_immune(k=k)

def is_affine(self):
    
    Determines if this function is affine.
    Returns:
        True if this function is affine, false otherwise.
    
    return True if self.nonlinearity() == 0 else False

def get_orbit(self, perms=None):
    
    Gets the orbit of this function under action of the symmetric group.
    Args:
        perms – default None. Uses this as a permutation set, otherwise
        the
        full symmetric group on n symbols.
    Returns:
        A list containing all functions in the orbit of this function.
    
    return orbit_polynomial(self, perms)

def nonlinearity(self):
    
    Gets the nonlinearity of this boolean function.
    
    Returns:
        int: Nonlinearity of this boolean function.
    
    return int(2**(self.n-1) - 0.5*self.walsh_spectrum())

def linear_structures(self):
    
    Creates a set of values that exist as linear structures of this
    polynomial.
    Returns:
        set: Set of linear structures.

    
    flatten = lambda l: [item for sublist in l for item in sublist]
    linear structs = set(flatten(self.listform))
    for monomial in self.listform:
        if len(monomial) > 1:
            linear structs -= set(monomial)

    return linear structs

def apply_permutation(self, perm):
    
    

Applies a permutation to the ordering of the variables to this function.

Args:
perm – The permutation to apply.

Returns:
The newly permuted function.

```python
def apply_permutation(perm)
    f = super().apply_permutation(perm)
    return BooleanFunction(f.listform, f.n)
```

```python
def __str__(self):
    return self.tex_str()
```

```python
def __add__(a, b):
    sum_f = FieldFunction.__add__(a, b)
    return BooleanFunction(sum_f.listform, sum_f.n)
```

```python
def __mul__(a, b):
    prod_f = FieldFunction.__mul__(a, b)
    return BooleanFunction(prod_f.listform, prod_f.n)
```

```python
def __eq__(self, poly2):
    return self.tableform == poly2.tableform
```

```python
def __repr__(self):
    return "BooleanFunction(%s, %s)" % (str(self.listform), str(self.n))
```

```python
def update_rule_table(self):
    rule_table_length = 2**self.n
    rule_table = [0]*rule_table_length
    for k in range(rule_table_length):
        point_to_evaluate = _dec_to_bin(k, self.n)
        rule_table[k] = self(*point_to_evaluate)
    self.tableform = rule_table
```

```python
def __hash__(self):
    return _bin_to_dec(self.tableform)
```

def get_x(n, field=GF2):
    """
    Gets a list of all possible x_i in order, from 0 to n-1.
    """
    if field == GF2:
        return [BooleanFunction([[i]], i+1) for i in range(0, n)]
    else:
        return [FieldFunction([[i]], i+1, field) for i in range(0, n)]

```python
def _gen_atomic(n, pos):
    prod = BooleanFunction([[GF2.get(1)]], n)
    for position, val in enumerate(_dec_to_bin(pos, n)):
        if val == 1:
            f = BooleanFunction([[position]], n)
            prod *= f
```
else:
    f = BooleanFunction([[position], [GF2.get(1)]], n)
    prod += f
    if prod.tableform[pos] != 1:
        raise BaseException("gen_atomic failed! Please report on Github!")
    return prod

def _GF2_to_ints(lst):
    return [1 if x == GF2.get(1) else 0 for x in lst]

def generate_function(rule_no, n):
    endFunc = BooleanFunction([], n)
    binary_list = _dec_to_bin(rule_no, 2**n)
    for pos, val in enumerate(binary_list[::-1]):
        if val == 1:
            endFunc += _gen_atomic(n, pos)
    return endFunc

def _bin_to_dec(num):
    ""
    Converts a binary vector to a decimal number.
    ""
    return sum([num[i]*2**i for i in range(len(num))])

def _dec_to_bin(num, nbits):
    ""
    Creates a binary vector of length nbits from a number.
    ""
    new_num = num
    bin = []
    for j in range(nbits):
        current_bin_mark = 2**(nbits-1-j)
        if (new_num >= current_bin_mark):
            bin.append(1)
            new_num = new_num - current_bin_mark
        else:
            bin.append(0)
    return bin

def _dec_to_base(num, nbits, base):
    ""
    Creates a binary vector of length nbits from a number.
    ""
    new_num = num
    bin = []
    for j in range(nbits):
        current_bin_mark = base**(nbits-1-j)
        if (new_num >= current_bin_mark):
            bin.append(1)
            new_num = new_num - current_bin_mark
        else:
A. Penland and W. Rogers

bin.append(0)
return bin

def _delta(x, y):
    """
    Returns 1 if x and y differ, 0 otherwise.
    """
    return x != y

def _hausdorff_distance_point(a, B):
    """
    Calculates the minimum distance between function a and the functions in
    the set B.
    """
    return min([a.hamming_distance(b) for b in B])

def _hausdorff_semidistance_set(A, B):
    return max([_hausdorff_distance_point(a, B) for a in A])

def hausdorff_distance(X, Y):
    """
    Calculates the Hausdorff distance between two sets of boolean functions.
    """
    HD1 = _hausdorff_semidistance_set(X, Y)
    HD2 = _hausdorff_semidistance_set(Y, X)
    return max([HD1, HD2])

def dot_product(u, v):
    """
    Basic mod 2 dot product.
    """
    s = sum(u[k] * v[k] for k in range(len(u)))
    return s % 2

def weight_k_vectors(k, nbits):
    """
    Generates all vectors with hamming weight k.
    """
    nums = range(nbits)
    vector_set_to_return = [
        vec_to_add = [int(y in j) for y in range(nbits)]
        for j in k_combinations:
            vec_to_add = [int(y in j) for y in range(nbits)]
            vector_set_to_return.append(vec_to_add)
        return vector_set_to_return

def weight_k_or_less_vectors(k, nbits):
    """
    Generates all vectors of weight k on nbits bits.
    Args:
        k – weight
        nbits – the number of bits
    Returns:
        All vectors of weight k on nbits bits.
```python
output = []
for i in range(0, k+1):
    output += weight_k_vectors(i, nbits)
return output

def _product(x):
    return reduce((lambda y, z: y*z), x)

def duplicate_free_list_polynomials(list_of_polys):
    """
    Takes a list of boolean functions and generates a duplicate free list of polynomials.
    """
    # Arguments
    list_of_polys (BooleanFunction): A list of polynomials.
    # Returns
    list: A duplicate free list of functions
    outlist = []
    for poly in list_of_polys:
        if True not in [poly == poly_in_out for poly_in_out in outlist]:
            outlist.append(poly)
    return outlist

def orbit_polynomial(polynomial, permset=None):
    """
    Orbits a polynomial using the given permutation set.
    Args:
    permset: A set of permutations to apply to the function
    Returns:
    A list of the polynomials created by the given orbits.
    """
    if permset is None:
        permset = Sym(polynomial.n)
    return duplicate_free_list_polynomials([polynomial.apply_permutation(i) for i in permset])

def orbit_polynomial_list(polynomial_list, permset=None):
    """
    Orbits a list of polynomials using the given permutation set.
    Returns:
    A list of lists of the polynomials created by the given orbits.
    """
    return [orbit_polynomial(polynomial, permset) for polynomial in polynomial_list]

def siegenthaler_combination(f1, f2, new_var):
    """
```

Generates a Siegenthaler Combination of two functions.

Args:
  f1 (BooleanFunction): The first function
  f2 (BooleanFunction): The second function
  new_var (int): New variable for the combined function.

Returns:
The Siegenthaler combination of $f_1$ and $f_2$

```python
f1_times_new_var = f1 * new_var
f2_times_one = f2
f2_times_new_var = f2 * new_var
return f1_times_new_var + f2_times_one + f2_times_new_var
```

def generate_all_siegenthaler_combinations(func_list, new_var):
    """
    Generates all of the possible Siegenthaler combinations
    of the given functions.
    
    Args:
    func_list – A list of functions to perform the Siegenthaler
    combination function on.
    
    Returns:
    A list of all possible Siegenthaler combinations for the given
    functions.
    """
    all_siegenthaler_combinations = []
    for f1 in func_list:
        for f2 in func_list:
            f1f2siegenthalercombination = siegenthaler_combination(f1, f2, new_var)
            all_siegenthaler_combinations.append(f1f2siegenthalercombination)
    return all_siegenthaler_combinations

def min_nonzero_dist(poly1, classA):
    """
    Determines the minimum nonzero distance between a polynomial and its
    nearest neighbor.
    
    Args:
    poly1 – A boolean function
    classA – A class of boolean functions.
    
    Returns:
    The minimum nonzero distance between poly1 and every element of classA
    .
    """
    dists = [poly1.hamming_distance(f) for f in classA]
    min_nonzero = float("inf")
    for dist in dists:
        if dist != 0 and dist < min_nonzero:
```
min_nonzero = dist

return dist

def reduce_to_orbits(f_list, permset):
    """Reduces a list of functions to a list of function classes given a permutation set.
    """
    basic_polys = []
    flatten = lambda l: [item for sublist in l for item in sublist]
    for f in f_list:
        if f not in flatten([orbit_polynomial(permset, basic) for basic in basic_polys]):
            basic_polys.append(f)
    return basic_polys
```