The Network Security Simulator, *Puissec*

Richard Pham

June 2024

Contents

1 Introduction

Puissec is a lightweight simulator that serves as a gauge (of quantities and categories) on the security of a network, with sensitive information, that is supposed to guard against 3rd-party intrusion. It is labelled as lightweight due to the minimal training data used by machine-learning needed to begin using it as a tool used to gauge the strength of security in an encrypted network. The name *Puissec* is a derivation of the words "poised","poison", and "security". Data poisoning consists of a body of techniques and theorem to contaminate empirical data in ways that result in mappings (i.e. transformations, conversions) with poor accuracy, typically measured by comparison of expected and actual output values. The theme of data poisoning is an optional feature in *Puissec* simulations. The simulator contains a few modes that will be gradually explained throughout this paper.

Puissec should not be thought of as an all-inclusive abstraction of network security operations in computer cracking. To clarify, the encyclopedia *Britannica* defines a computer simulation as a body that "represents the dynamic responses of one system by the behaviour of another system modeled after it."[Csi] Automaton models were conceived to help generalize network security operations as functions and decision problems for computer agents of *Puissec*. These structures run by processes that have arbitrary or shallow qualities in likeness to real-life encryption, such as Data Encryption Standard.[Lec]

The data structures in *Puissec* were designed with the intent to emulate key characteristics of real-life cracking operations. Said key characteristics include the usage of brute-force attempts, in common iteration patterns using templated data, to "crack" protected information, compartmentalization of information (typically through segmenting components of the entire information into nodes of a network with variable levels of connectivity), interception of messages that are protected at endpoints but where transmission between the endpoints remains vulnerable, and induction techniques used to improve the comparison of hypotheses with actual values (so that brute-force runtime can be decreased), that which concerns components of the protected information.

A last key characteristic of real-life cracking operations is that effective intruders have a relevant degree of background information on the target to be cracked. This background information, in the realm of cryptography, includes variables such as the knowledge of the encrypting and decrypting algorithms, auxiliary information such as public/private keys, and probability maps for generating and verifying hypotheses on the encrypted information. The variety of cryptographic algorithms is difficult to represent in a generic form. Cryptography's proprietary/secretive nature adds more to the difficulty of collating the various cryptography algorithms into a lightweight form with no need for a backend engine that utilizes a database of cryptographic algorithms. To keep software operations simple, *Puissec* represents secretive information as containers with one "private" variable, and the other variables typically marked "public". Third-party agents that attempt unauthorized access for the knowledge and/or possession of the "private" variable will have to conduct guesses based on their hypotheses and stochastic decision-making with the consideration of the "public" information of the other variables.In the operational sense of networks, effective intruders tend to possess geometric knowledge and competency, that is, effective navigation and other examples of decision-making so that intrusion attempts go smoothly and not in roundabout ways which increase probability of anomaly detection by the security network. Network calculations (functions operated over elements of a graph) are implemented in *Puissec* to aid in efficient agent navigation.

Decisions taken by agents that involve selecting arbitrary elements out of possible candidates use **pseudo-random number generators** that output indices for selection. Machine-learning, a term that generally refers to the broad body of theoretical mathematics and computational techniques used to calculate fitting solutions by use of training data, is not a focus of the research and development of *Puissec*. For simple cases of network security analyses, running simulations on the same structure containing the secret using multiple pseudo-random number generators (that have a degree of variability between each other above a certain threshold value) should produce complete probability distributions on performance measures of a third-party agent's unauthorized attempts against a security network.

2 To Represent a Secret

Considering the fact that *Puissec* is a computer program that "simulates" the activity between an unauthorized third-party and a network that contains a "secret" that it is supposed to guard access (knowledge and/or possession) against the unauthorized third-party (commonly called a "cracker" or "intruder" in computer security lingo), the program uses the simple structure that is a vector V, such that V satisifes the condition $V \subseteq \mathbb{R}^{|V|}$, as the basic unit (the smallest structure as the building block) to construct a representation of a secret. This preference for vectors comprised of real numbers is due to two main reasons.

- large (quantitative) and diverse (qualitative) space,
- compatibility with standard mathematical operations, as opposed to more technicalities with real-world cryptographic operations, such as the complexity behind single sign-on operations in modern internet security.

The vector serves as a balance between single float/complex numbers and numerical structures with greater geometric complexity (i.e. prisms, multi-dimensional matrices). Consider the field of computer cracking, techniques used to gain unauthorized access into computer systems (such as through bypassing authentication protocols, encryption, et cetera).[Com] An arbitrary vector V of an arbitrary dimension containing elements of $\mathbb{R}$ has a difficulty of being correctly "guessed" (cracked) that corresponds to the dimension and the subspaces of $\mathbb{R}$ that are searched.

2.1 *Sec* as the Unit

The class used to represent a portion of a secret is named *Sec*. Each *Sec* instance used to comprise the entire secret can be supposed as a separate compartment. The term "compartment" is used to denote an enclosure in the whole, such that the compromise of that enclosure has a relation to the compromise of other parts of the whole,of a degree below an arbitrary small threshold. In other words, the design of the structure for an entire secret as a set of *Sec* instances, each considered a compartment, is to produce a fault-tolerant defense by simulation agents that are assigned the task of given the objective of defending relevant information from the secret from third-party access and/or retrieval.

Fault tolerance is a process that enables an operating system to respond to a failure in hardware or software. This fault-tolerance definition refers to the system's ability to continue operating despite failures or malfunctions [**ftb**], and is applied as a concept to the design of a secret as a conglomerate of *Sec* instances.

The following is a list of its variables.

- V, of arbitrary dimension in $\mathbb{R}$.
- P_O, the optima probability map
$$\texttt{optima id} \rightarrow v \in (0., 1.].$$
- M_d, the dependency map connecting the *Sec* to other *Sec* instances,
$$\texttt{dependency identifier } \rightarrow r \in (0., 1.].$$
- M_c, the co-dependency map connecting the *Sec* to other *Sec* instances,
$$\texttt{co-dependency identifier } \rightarrow r \in (0., 1.].$$
- F_b, a "blooming" function that produces false copies of the *Sec*.

The vector V of a *Sec* instance S is its true form, and is the only required variable of the structure to be kept private (outside of the certain knowledge of a third-party agent). This is the sequence of float values that is the authentic information a third- party agent wants access to. But there are, in total, $|P_O|$ credible candidates for the true form of S, in the space of real numbers with dimension $|V|$. **A credible candidate for $S.V$ corresponds to a probability value that matches the expectations by the perspective of the third-party agent's background knowledge.** This design serves as a simple and fallible first defensive barrier against the third-party agent's knowledge of $S.V$ in the form of a probability distribution for candidates.

For the maps M_d and M_c, the probability values pertain to the strength of connection (other words are "bond" and "association") between two local optima, each belonging to a separate *Sec* instance. Both the maps M_d and M_c

have keys that each consist of four values: the first two denote the *Sec* identifier and its local optima index, and the second two are those for the other *Sec* instance. The terms "dependency" and "co-dependency" pertain to an ordering termed in this paper as the *order-of-cracking*. This is an ordering for attempting to access the true form of *Sec* instances. Each element in this ordering is a set of *Sec* instances. For elements that consist of more than one *Sec* instance, the *Sec* instances are co-dependent. A third-party agent will have to attempt to access all of these *Sec* instances at once. To elaborate, suppose that a third-party agent intends to attempt to access local optimum i belonging to *Sec* j. But the map M_c shows codependencies of (i, j) with the set $\mathbb{Y}$ of elements,

$$\mathbb{Y} = \{(k_i, k_j) : k_i \texttt{ a } \textit{Sec} \texttt{ not equal to i, } k_j \texttt{ a local optima of } k_i.\}$$

In order to attempt to crack (i, j), the third-party agent has to also attempt to crack all elements of $\mathbb{Y}$. If this third-party agent wants to attempt access to (i, j), but the map M_d lists the set of elements

$$\mathbb{Y}_{\not\models} = \{(k_i, k_j) : k_i \texttt{ a } \textit{Sec} \texttt{ not equal to i, } k_j \texttt{ a local optima of } k_i\}$$

as required dependencies, then the third-party agent can proceed to attempt access to (i, j) if and only if all elements of $\mathbb{Y}_{\not\models}$ have already been accessed.

The dependency map shows the strength of connection for each of its existing keys , a pair of (*Sec* identifier, local optimum index) values, but that which differs from co-dependent connections. For two secrets' local optima I_0,I_1 with an existing connection found in M_d as "I_0, I_1", then I_1 needs to be attempted before I_0. In other words, the attempt to access I_1 depends on the successful attempt to access I_0.

The keys of the maps M_d and M_c can be thought of as boolean values for the existence of connections between two (*Sec* identity,local optimum) pairs g_1, g_2.For a key k defining a connection between g_1 and g_2, there is the condition of co-dependent attempts to access or attempts that depend on previous access attempts to those pertinent and existing keys.

There are some conditions for establishing dependent or co-dependent connections between two local optima belonging to different *Sec* instances. For a *Sec* S to be able to be dependent on another *Sec* S_2, the condition that S_2 is not dependent on S must hold. There is also the condition that S is not co-dependent with S_2. And if S were co-dependent with S_2, then there cannot be any dependent connection between S and S_2.

The precise probability values that make up the output space of these two maps are used in linear combinative mappings involving the probability values of local optima (the map L_O for each *Sec*) to produce new probability values. These values will be discussed in fuller context later.

2.2 The Secret as Comprised of *Sec*

Building on top of this introductory description to the *Sec* structure, this paper proceeds to the representation of an entire secret. The true form of any *Sec* instance is a vector of arbitrary dimension in $\mathbb{R}$. Below is the abstraction of a secret in *Puissec*.

Definition 2.1 (Secret (in *Puissec*))**.** A data structure intended to represent a secret through a set of "particles" (each a *Sec* instance), such that each "particle" contains the true contents of a portion of the secret and may be connected to other *Sec* instances through co-dependencies and dependencies, attributes related to corresponding probability value outputs and the order by which third parties attempt to access specific *Sec* instances.

By the above construct, it is relatively convenient to map (i.e. translate) a sequence of characters, and this is the typical format of a communicative object such as a message, to a sequence of values in $\mathbb{R}$.

2.3 Implementing a Defense Mechanism by the Design of *IsoRing*

An *IsoRing* is a data structure that encapsulates a *Sec* instance. In the theoretical design for *IsoRing*, the structure does not allow third-party agents to access any of the variables V,P_O,M_d,M_c, and F_b. However, the first version of *Puissec* is written in Python, a high-level programming language without any built-in public/private variable access.[Pri]

There are first some clarifications to be made on the design of the attempt-to-access procedure.To guess the true form of a *Sec* instance S, a third-party provides a tuple of two elements, that is the identifier for the local optima (typically an integer in *Puissec*) and a vector V' that is supposed to equal $S.V$, the true form of S. The program enacts the assumption that the third-party has knowledge of the dimension of $S.V$ for consideration of the guess. The attempts as vector V' are referenced against $S.V$ to produce scores outputted back to the third-party agent for it to gauge its performance. This referential function, F_r, is typically the standard Euclidean distance between two vectors of the same dimension,

$$\mathbb{D}(V_0, V_1) = \sqrt{\sum_{i=0}^{|V_0|-1} |V_0[i] - V_1[i]|^2}.$$

Puissec allows for more output variation by using other functions beside from the standard Euclidean distance. The function $\mathbb{D}$ is the most accurate out of the possible three functions (the other two have not yet been detailed) that can be used to output a **distance score** back to the third-party agent. This distance score is used as an important indicator variable, important because it is one of the few values transmitted back to a third-party agent for each of its attempts to access a *Sec* instance in the 2-tuple format previously defined. If the function $\mathbb{D}$ is the only one used to output the distance score, then the third-party agent has an accurate feedback variable that it can use to improve on its guesses. More specifically, the third-party agent has an accurate conception of the travelling directions along the dimensional space of V, the true form of the targeted *Sec* instance, to take so that the distance score is minimized to the value of 0.0. In such simple cases, the third-party agent reasonably assumes its guess is the correct value of the specific local optimum index belonging to the targeted *Sec* instance.

2.3.1 Available Distance Functions in *Puissec*

The other two functions are named **scalar modulo**, $\mathbb{D}_{\nvDash}$, and **random noise**,$\mathbb{D}_{\nVdash}$. Both of these functions take the same arguments as $\mathbb{D}$, and also use $\mathbb{D}$ to produce an intermediary distance value d for the two arbitrary vectors V_0 and V_1. For the function $\mathbb{D}_{\nvDash}$, a non-negative integer n is multiplied with d to produce the output $d \times n$ as the distance score back to the third-party agent. The source that generates these non-negative integers, used as multipliers, is a *pseudo-random number generator.* The definition provided next is essentially a slight rephrasing of that found in its source (see cited).

Definition 2.2 (Pseudo-random number generator.)**.** A deterministic polynomial-time algorithm G is a pseudo-random number generator if $\exists$ a stretch function,

$$L : \overrightarrow{I} \to \overrightarrow{O},$$

$$\mathbb{S}^{|\overrightarrow{I}|} \to \mathbb{S}^{|\overrightarrow{O}|}$$

such that $|\overrightarrow{I}| < |\overrightarrow{O}|$. Typically, the number set $\mathbb{S}$ is one of $\{\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{C}, \mathbb{Q}\}$. Suppose for any arbitrary distinguisher function F_d that takes as input a sequence of elements in $\mathbb{S}$, F_d is able to classify (distinguish) selections of elements belonging to $\mathbb{S}$ as certain labels. In generic cases, this labelling is the output space $\{0, 1\}$. Then for any positive polynomial p, sufficiently large k, and any pair of values $(\overrightarrow{I}, L(\overrightarrow{I}))$ such that $\overrightarrow{I} \in \mathbb{S}^{|I|}$ and $L(I) = \overrightarrow{O} \in \mathbb{S}^{|L(I)|}$, it holds that

$$|Pr[F_d(G(\overrightarrow{I})) = 1] - Pr[F_d(G(L(\overrightarrow{I})) = 1]| < \frac{1}{p(k)}.$$

Given the output from the generator function G, the difference between the probability that F_d registers $G(\overrightarrow{I})$ as 1 (abnormal) and that where F_d registers $G(L(\overrightarrow{I}))$ as 1 is virtually equal. [Gol10]

The pseudo-random number generator G preserves the distinguishability of qualities between samples of the input space and their correspondents in the output space. This definition contains areas of interpretation, especially on the term "distinguishable". The textbook that served as the source for this definition elaborates on the concept of pseudo-random generators through theoretical prose. In machine-learning applications, the calculation of "distinguishing" the qualities of a sample in order to compare with other samples rests on functions pre-trained and fitted according to relevant data. There is an implementation of a structure in *Puissec* that is a "weak distinguisher", used by the third-party agent, that will be described alongside formal description of a third-party agent in the context of *Puissec.* The focus of this paper does not require any more in-depth exemplification on the aspects of pseudo-random generators. But any extension of *Puissec*'s existing pseudo-random number generator does require at least an awareness of the theoretical construct that is a generator outputting "pseudo-random" values.

The **scalar modulo** function is defined as

$$\mathbb{D}_{\nvDash}(V_0, V_1) = \mathbb{D}(V_0, V_1) * F_p(),$$

$$F_p \texttt{ a function that outputs an integer in a range } [0., d].$$

The non-negative integer d is arbitrary selected, but a relatively small but complete range such as $[0, 10]$ suffices as the output space for multipliers applied to output values of function $\mathbb{D}$. The resulting value has a high probability of being an inaccurate distance value between two vectors V_0,V_1. The recipient of these values, the third-party agent, will have to resort to other methodologies in determining the spatial relevance (minimal distance) of its guess to the local optima of a *Sec* instance it is targeting. And the function F_p is a pseudo-random number generator that has already been instantiated with a "seed" value, and outputs an integer value every time it is called.

The function **random noise** is similar to **scalar modulo**, except the operator is $+$ instead of $*$ and it uses the pseudo-random number generator $F_p^{\mathbb{R}}$, which outputs values in a subspace of $\mathbb{R}$ instead of $\mathbb{N}$:

$$\mathbb{D}_{\nVdash}(V_0, V_1) = \mathbb{D}(V_0, V_1) + F_p^{\mathbb{R}}(\dots).$$

2.3.2 The *BoundedObjFunc* Functional Structure

The *IsoRing* structure is not constrained to using any one of these three distance functions. Instead, it uses a structure called a *BoundedObjFunc*, the codified spelling for "bounded objective function". The *BoundedObjFunc* $\mathbb{B}$ is essentially a container for an arbitrary number of distance functions, usually kept under an arbitrary small value such as ten for timely processing of bulk sequences of vectors and related information that comprise brute-force guesses. The structure takes two sequences, each with an equal number of elements to the other.

- Q_0: a sequence of function instances, each one an instance of one of the categories in $\{\mathbb{D}, \mathbb{D}_{\nvDash}, \mathbb{D}_{\nVdash}\}$.
- Q_1: sequence of elements, each an $(n, 2)$-dimensional range matrix, otherwise known as a bound. The non-negative integer n is equal to the length of V, the vector and true form of the associated *Sec* instance, and each row is ordered. For any two bounds $b_0, b_1 \in Q_1$, and for any index d' such that $d' < d$, there cannot be any overlap between $b_0[d']$ and $b_1[d']$. The below function outputs the boolean of no overlap between any two bounds along a specific dimension.

```
function $NO\_OVERLAPS(b_0,b_1,i)$
    if $b_0[i,0] \geq b_1[i,0]$ & $b_0[i,0] \leq b_1[i,1]$ then
        return False
    end if
    if $b_0[i,1] \geq b_1[i,0]$ & $b_0[i,1] \leq b_1[i,1]$ then
        return False
    end if
    return True
end function
```

The qualities of specific *BoundedObjFunc* instances are explored using a type of algorithm called a "weak learner", to be explained in a later section.

2.4 Probability-Value Functions

Each probability value of the map L_O is the odds of its corresponding key, a vector in the same dimension as V, being the most fit. The summation of the probability values equals 1.0. In typical cases, V has the greatest probability, but this is not a required condition.

There are four categories of probability-value functions, each accepting a pair of values (*Sec* identitifier,local optimum index) as input.

- P_o (local optimum probability function)

  the probability-value function that outputs the probability value of a vector V' if $V' \in L_O$. If $V' \notin L_O$, then output 0.0. It accepts a pair of values (*Sec* identifier, local optimum index).

- P_d (dependent probability function)

  Suppose to attempt access to optimum i belonging to *Sec* j, the elements of this set were taken as decisions in the previous duration,

  $$S_D = \{(\ \texttt{\textit{Sec} identifier, local optimum index):}$$

  $$\texttt{the element must be accessed before accessing i for j}\}.$$

  Then the dependent probability function on optimum i of j first applies a delta to the value of $P_o(i,j)$ by the function,

  $$P'_d(i,j) = S_j.L_O[i] + \sum_{x \in S_D} (S_j.L_O[i] * M_d[i,j,x[0],x[1]]).$$

  Set another probability map $L_O^{(1)}$ as equal to L_O for all (key,value) pairs except for optimum i, with the new value equal to $P'_d(i,j)$. Then the new probability value for each local optima $V \in L_O$ is

  $$P_d(i,j) = L_0^{(1)}[i] / \sum_{V \in L_0^{(1)}} (L_0^{(1)}[V])$$

- P_c(co-dependent probability function)

  The function P_c is virtually identical to the form of P_d except instead of the set S_D of decisions taken for dependent connections, it is the set S_C for co-dependent connections, which contain pairs denoting the local optimum of other *Sec* instances being attempted access at the time (co-occurrence) by the third-party agent, and instead of the map M_d, the map containing probability values for existent co-dependencies M_c is used.

- P_{cd} (weighted connection probability function)

  Given a 2-tuple $W = (w_0, w_1)$ that is a weight vector such that $w_0, w_1 \geq 0.0$, $w_0 + w_1 = 1.0$, and w_0 the weight for the output of P_d and w_1 the weight for the output of P_c, then

$$P_{cd}(i,j) = \frac{P_d(i,j) * w_0 + P_c(i,j) * w_1}{2.0}.$$

In the next section (2.3.1), there is discussion on the pertinence of these probability values as response variables fed back to the third-party agent during their attempt to access. These values can each be utilized as identifiers for their corresponding vector (a vector that is a local optimum index). This utility proves to be accurate in distinguishing corresponding local optimum/s in cases where the probability distribution of the set of local optima is completely non-uniform, meaning there does not exist any pair of local optima l_0, l_1 such that

$$Pr(l_0) \neq Pr(l_1).$$

2.5 Probability-Value Fitting

A third-party agent uses the concept of **probability-value fitting** as one aspect to verify the authenticity of an attempt, the authenticity of its guess V' being equal to the true form of its targeted *Sec* instance. The distance functions have already been discussed as devices used to output distance scores (a category of response variable) back to a third-party agent for every one of its guesses to access. If a third-party agent happens to provide a guess (i, V') such that V' is the correct value of the local optimum at index i, then it receives another response variable belonging to a different category, the probability value of the guess. The third-party agent compares this received probability value with another in what is termed **probability-value fitting**.

Definition 2.3 (probability-value fitting)**.** In the program *Puissec*, a third-party agent receives probability values on the authenticity of every attempt to access a local optimum of a *Sec* S. The third-party agent then compares these probability values with its hypothesized probability value for the true form of S. In cases where there are numerous (many more than one) guesses that received a probability value equal to the third-party agent's hypothesized probability value, then the odds that the third-party agent can determine that its guess equals the value of the true form of S instead of that of one of S's differing local optima is probabilistic instead of absolute.

The concept of *probability-value fitting* is an application of information science in the context of multi-agent knowledge bases. There is a specialized data structure in *Puissec* that represents the hypotheses by the third-party agent. It is named **BackgroundInfo** in the programming code of *Puissec*, and is discussed in a later section that describes the design and capabilities of the third-party agent.

2.6 Gathering Probability Values Into Categorical Prisms

There is a structure coded *SRefMap* that helps to calculate a set of maps pertaining to decisions and probability values, based on the input. The input consists of a set of *Sec* instances that constitute a secret. The maps L_O,M_d, and M_c defined for the *Sec* structure are used extensively as variables by *SRefMap*.

And for the terms "decisions" and "probability values", they pertain to *Puissec* specifications. Recall the structure of a third-party agent's guess as

$$(\texttt{\textit{Sec} identifier, local optimum index}).$$

There is the decision of a third-party agent on which of the optima indices to guess in its attempts-to-access a targeted *Sec* S_i instance. By default, as in a uniform probability distribution, every local optima $l \in L_O$ has an equal and non-zero chance of being selected. The probability values, outputted in formats involving real numbers in the range $[0., 1.]$., are calculated by this data structure to serve as one response value back to a third-party agent with the guess (i, j) that satisfies the condition

$$S_i.V \overset{?}{=} j.$$

For the true form of a secret as a set of *Sec* instances, there is a map termed a **decision map**,

$$M_{dec} : \texttt{\textit{Sec} identifier} \rightarrow \texttt{integer index of local optimum}.$$

Every secret is associated with a decision map, and the values of this structure is not public knowledge to any third-party agent.

The available modalities for *SRefMap* to output probability and decision maps consists of this listing.

- 1. decision map functions: one of co-dependent,dependent, or weighted connection.
- 2. probability-value functions: one of lone,co-dependent, dependent, or weighted connection.
- 3. extremum function: minimum or maximum.

The term "prism" in the context of *SRefMap* is a **decision map** and a corresponding **probability value map** associated with the three-tuple that is a descriptor for its modes. The uniqueness of the $3 * 4 * 2$ different prisms is dependent on the number and geometric relevance of the connections denoted by maps M_c and M_d. Null maps result in identical (decision,probability) map pairs between the prisms of different modes.

SRefMap first calculates a map M_x of the form

$$M_x : \texttt{Sec identifier} \rightarrow \texttt{local optimum index}$$

$$\rightarrow (\min(M_{dec} \in SPACE(M_{dec}), \max(M_{dec} \in SPACE(M_{dec})))$$

by iteration of the *Sec* instances belonging to a secret. For full sampling onto a map, all available local optima indices for each *Sec* is processed. The second output layer of M_x is an ordered pair of decision maps. The first map contains the decisions to be taken for minimizing the probability value output associated with the key (the j-th local optimum belonging to the i-th). The second map is the decisions for maximizing the probability value output. There are three possible maps that can be M_x (with no observation of their uniqueness), each one by one of the decision map functions (co-dependent,dependent,weighted).

For the probability map values, lone probability values outputted for correct (*Sec* identifier, local optimum index) instances are the direct values in the map L_O of the *Sec* i. For the other types of probability maps, they utilize the functions P_c,P_d, and P_{cd} that were defined in section 2.4 (Probability-Value Functions).

After the map M_{dec} has been calculated based on parameter #1 (decision-map function type) x, an input x_2 (the value for parameter #3, respectively) obtains a decision map $M_d^{(0)}$ for one of the prism's variables. Then to obtain the probability map, use the local optima decision map $M_d^{(0)}$ as the set S_D defined in the definition of P_d. Then the probability values in the output space of the output probability map are calculated according to the function designated by parameter #2.

Prisms are calculated by first setting parameter #1. Then *SRefMap* calculates prisms for all possible selections of parameters #2 and #3. There are $4 * 2$ prisms (some may be not unique) for each selection of parameter #1, for a total of $3 * 4 * 2$ total prisms.

2.7 Accounting for Feedback-Response Loops in Real-World Situations

The previous section first defined the format of a **guess** by a third-party agent as a 2-tuple,

```
(local optima index, vector value).
```

The general procedure, a feedback-response process, consists of these steps.

- A third-party agent T is attempting to access local optimum index j belonging to *Sec* S_i.
- T produces the guess G, and S_i receives G.

- S_i transmits the distance score of G using its private distance function $\mathbb{D}_i$.
- Additionally, if the guess G is correct, then S_i transmits a probability value on the authenticity of G being the true form of S_i.

This feedback-response loop is an attempt to abstract the typical course of events a third-party agent attempts to bypass protection to access an object, in which the observable reaction (the probability value) from the attempt is transmitted back to the third-party agent. There are a few classifications for attempts-to-access.

All attempts-to-access in *Puissec* are unauthorized. An attempt (l_i,v) is categorized as a **successful attempt** if for the i'th optimum of the targeted *Sec* S, $S.L_O[i] = v$, although the true form of S that is the optimum at the j'th index does not have to satisfy $j = i$. If additionally, j equals i, then the **successful attempt** is also a **crack**.

2.8 Obscurantism by IsoRing

The *IsoRing* instance I performs an obscurantist operation on the central *Sec* instance S it encapsulates. By using the function F_b, the blooming function, I produces x additional "false copies" of S.

Definition 2.4 (False Copy (of *Sec*))**.** With respect to a central *Sec* S that is the source of its bloom generator function F_b, a false copy S' has a true form V' such that $|S.V| \neq |V'|$. The differing dimensions between S and S' is why S' is a false copy.

Obscurantism is defined as "opposition to the spread of knowledge: a policy of withholding knowledge from the general public."[Obs]. To exemplify, to obscure something is to preserve the form (or existence) while projecting a different image from the most revealing (the complete truth). For example, dogs are canines, so the category of "dog" is a subset of "canine". To use the word "canine" in place of "dog" would obscure the true meaning of the original message that uses the word "dog" since the superset is not as specific. By a basic calculation, the word "canine" could obscure 0.5 of the meaning of the original word "dog".

To generate the next *Sec* instance (before termination), the blooming function F_b outputs a dimension d that is not yet the dimension of any one of the copies associated with the central Sec instance. If there does not exist any more unique dimensions d outputted by F_b, then F_b ceases the generation of false copies for S. Otherwise, F_b generates an arbitrarily-sized L_O,M_d,and M_c for the false copy on a unique dimension d. For best practices, the blooming function should output these maps according to a range of sizes that result in a uniform size distribution across dimensions, each dimension associated with a copy of S. All copies of a Sec instance generated by F_b are false copies. An *IsoRing* holds the false copies and the central Sec. At any single point in time during a network security simulation, the "projection" of an *IsoRing* to a third-party agent is a singular focus on one copy of the *Sec S* the *IsoRing* encapsulates as the representation available to a third-party agent for access. There is one condition required to be met for an *IsoRing* to switch to another copy for projection. But the details are placed in a future section on the logistics of a third-party agent conducting unauthorized attempts to access. The specific probability values for each generated false copy of S are programmatically derived through two classes of probability derivation: **exact** and **partial** correlation.

2.8.1 Probability-Value Calculations For False Copies

The structure used to "bloom" (derive,generate) a *Sec* instance into another of a different dimension is called *OptimaBloomFuncSecRep*, and this structure in turn has two structures *DerivatorPrMap* and *OptimaBloomFunc*.

Remark. Not all the attributes from these structures are fully described in this writing.

DerivatorPrMap is used to calculate probability values for lone optima as well as dependent and co-dependent connections. *OptimaBloomFunc* is used to generate the local optima for the bloomed *Sec*. Broadly speaking, the decisions that it makes are used by *DerivatorPrMap* to output a probability value for every input it receives. A number generator in two-dimensional space, called *Index2DMod* by the program *Puissec*, outputs two-dimensional values in $\mathbb{N}_{\geq 0}$. Outputs from *Index2DMod* are used in pairs, as indices i_1 and i_2, to select two operands to be applied by a pairwise operator, $*$. The pairwise operator is typically one of the four basic arithmetic functions. Given a reference *Sec* instance S that may or may not be the source, select a new dimension d and a local optima size i_l. Then instances *DerivatorPrMap* P_{pr} and *OptimaBloomFunc* P_b are declared. The *OptimaBloomFunc* uses *Index2DMod* to output index vectors $\overrightarrow{i}_1$, $\overrightarrow{i}_2$, each of length equal to d. A **counter** C_T (essentially a map) is used to record these index vectors. The terms **keys** and **values** are used to refer to the input elements (not the entire possible space) and output elements.

Specifically, an external process provides an axis of counting, an element $a \in \{0, 1\}$. Then to update C_T from time unit u to $u+1$, set the concatenation of $\overrightarrow{i}_1$ and $\overrightarrow{i}_2$ as $\overrightarrow{i}_3$.

$$C_{T_x}^{(u)} = C_T^{(u)}[i_x[a]] + +; i_x \in \overrightarrow{i}_1,$$
$$C_T^{(u+1)} = C_{T_x}^{(u)}[i_x[a]] + +; i_x \in \overrightarrow{i}_2.$$

The counter C_T as used in blooming functions is instantiated as empty at time unit u. After it counts the indices $\overrightarrow{i}_1, \overrightarrow{i}_2$ of the reference *Sec S*, P_{pr} proceeds by one of **exact** or **partial** probability-value calculations.

If the mode is **exact**, fetch an element

$$k = \max_{q \in keys(C_T)} (C_T[q]).$$

Then the probability value for the new optimum of $\overrightarrow{i}_1 * \overrightarrow{i}_2$ and length d is $S.L_O[k]$.

If the mode is **partial**, the algorithm uses a weighted scheme to produce a value r in $[0., 1.]$:

$$r = \sum_{k \in keys(C_T)} (C_T[k] * S.L_O[k]) / \sum_{v \in values(C_T)} (v).$$

2.8.2 Summarization of *IsoRing*

- Uses the feedback-reponse process to transmit distance scores and probability-value scores.

  [-] The distance scores are calculated by an arbitrary instance of *BoundedObjFunc*.

  [-] The probability-value scores are preprogrammed directions given to the *IsoRing* by a regulatory structure called a *SecNet* (the security network).

  [-] The *SecNet* structure gives directions for *IsoRing*'s probability-value responses according to the *SRefMap* that processed the probability maps of the secret.

- Obscurantist process is a design that accounts for vulnerabilities through coincidences or parallelisms in interpretation of communication objects, such as textual messages. An *IsoRing* structure can construct an arbitrary number of copies for the central *Sec* it encapsulates using the blooming function F_B.

2.9 The Security Network, *SecNet*

The security network is instantiated with an input involving a sequence of *IsoRing* instances, each with a unique identifier. A *SecNet* instances accepts as one of its input variables (or generated variables) a simple, undirected graph.

Definition 2.5 (simple, undirected graph). A connective structure with a **node** as the unit for a point of connection and a **edge** as a bi-directional connection between any two non-identical nodes n_0, n_1. For an edge (n_0, n_1), the edge (n_1, n_0) is identical. There cannot exist more than one edge between any two nodes. For an arbitrary simple, undirected graph G, the set of nodes is $G.V$ and the set of edges is $G.E$.

There is a protected network structure that consists of a simple, undirected graph G, with two associated sets. One is designated protected (SEC) nodes and the other is unprotected ($NSEC$) nodes. Every node $v \in G.V$ is one that is exclusively SEC or $NSEC$. There is also a set of nodes that a third-party agent can use as "entry" points (ENT) into the security network. The only constraint for the size of ENT is $0 \leq |ENT| \leq |G.V|$. A node that is labelled SEC is a secured node of the network. On the contrary, an $NSEC$ node is not a secured node of the network. The ramifications of a node being a SEC

or $NSEC$ node are explained in the ending section (2.9.2) on the design of the graph generator templates alongside unauthorized third-party agent activity. The essential difference between SEC and $NSEC$ is that SEC nodes provide a level of security against third-party agent access. And thus, these nodes have more strategic value in the context of security networks, and in cases of equal economic valuation for SEC and $NSEC$ labels, SEC nodes are always preferred to satisfy threshold requirements for defense.

For every node of the simple, undirected graph pertaining to a *SecNet*, there can exist an arbitrary number of *IsoRing* instances located on it. At any instance in time during the running of a *SecNet*, any *IsoRing* instance must be located on exactly one node. There is one other condition in *Puissec* that need to be met in generating a possible protected network. Due to the concept of traffic-jamming manifesting itself in transportation systems, including graphs, there is the rule that the number of nodes belonging to G must be at least equal to the number of *IsoRing* instances. The process templates for generating a protected network through stochastic decisions (by a pseudo-random number generator) are **spine frame**, **pairing frame**, and **random frame**.

2.9.1 Graph Generator Templates

Each of the graph generator templates require input arguments specifying constraints for generating the graph (assume to be simple and undirected unless otherwise specified).

- N_{sec}, the set of nodes belonging to the SEC category.
- N_{nsec}, the set of nodes belonging to the NSEC category.

Additionally, the generator templates use a pseudo-random number generator (typically Python 3's standard random library) to aid in decisions on adding edges to the output graph G_O that starts off as a graph with the node-set $SEC + NSEC$ and a null edgeset.

All three generator templates output a simple, undirected graph G (with additional features pertinent to *Puissec*) that has one component. The following two definitions clarify on the definitions relevant to this condition.

Definition 2.6 (path (graph theory)). A path could be a simple,undirected graph G such that for any two non-identical nodes $n_0, n_1 \in G.V$, there exists a sequence of edges $\{e_0, e_1, e_2, \dots\} \subseteq G.E$ with a size between 1 and $|G.V| - 1$. Every node shares either one or two edges (degrees) with another node. There exists exactly two nodes with degree of one in graphs with number of nodes greater than one.

Definition 2.7 (component (graph theory)). A component C is a subset of nodes belonging to a simple, undirected graph G such that for any two nodes $c_0, c_1 \in C$, there exists a path $\overrightarrow{X}$. For every edge (n_0, n_1) in this sequence $\overrightarrow{X}$, $n_0, n_1 \in C$. For a node n_0 of C, all of its neighbors in G are included in the component.

The **spine frame** procedure starts out as an empty and mutable graph G. Then for the subset of nodes NSEC, a pseudo-random number generator is used to select the ordering of the nodes in NSEC for a path to be drawn between them to produce an updated version of G (the running solution).

Then for the remaining disconnected nodes that is also the set SEC, iterate through the nodes of SEC by an arbitrary ordering (with the use of a pseudo-random integer generator Q_i). For each iterated node n, an edge is drawn between n and a node of G (the running solution). The rule below is used in deciding on the node $n_x \in G.V$ to connect to n.

If $G.V \bigcap SEC = \emptyset$, then use Q_i to select one of the nodes that have degree one. Otherwise, choose the node n_x that maximizes the score output from the distance function d_f. An auxiliary function used by d_f is f_a. The auxiliary function takes a node n_x in SEC as an argument. It simulates a graph G' such that for the running solution, $G \subseteq G'$ and $G'.E - G.E = \{(n_x, n)\}$. Then for the auxiliary function f_a,

$$f_a(n_q) = \sum_{n_r \in SEC} s_{path}(G', n_q, n_r).$$

The function s_{path} produces the shortest integer edge-distance in the simple, undirected graph G' between two nodes n_q, n_r as its arguments. If the nodes n, n_q are not connected (there is no sequence of edges belonging to the graph G such that n and n_q each constitutes one of the nodes with degree one), then s_{path} outputs ∞.

The distance function d_f uses the function f_a as such:

$$d_f(SEC) = \max_{s \in SEC} f_a(s).$$

This procedure is for the case of a non-empty SEC nodeset. If $SEC = \emptyset$, then choose another arbitrary node n_x by Q_i, not connected to the graph of n, that is connected to n by an additional edge to the running solution G. Continue this process until the first instance that the running solution G has one component (a connected set of nodes) such that $G.V = SEC + NSEC$.

The **pairing frame** generator template is so named because of the nature by which it connects a set of nodes (each of either SEC or NSEC) into one connected component. There are two categories of edges that can be added to the running solution.

- inter-component connection: connects two components $C_0, C_1 \subseteq G.V$ by an edge
$$(n_0, n_1); n_0 \in C_0, n_1 \in C_1.$$
- intra-component connection: connects two nodes n_0, n_1 belonging to a component $C \subseteq G.V$ by adding (n_0, n_1) to $G.E$.

Instead of starting with an empty graph, this procedure starts as G, with $G.V = SEC + NSEC$ and $G.E = \emptyset$. For the first iterative cycle, conducted over each node $n_s \in SEC$, choose a node $n_c \in NSEC$ for an edge (n_s, n_c). The choice of $n_c \in SEC$ has the maximum score from the function $degree_{sec}$.

$$degree_{sec}(G, n) = |neighbors(G, n) - NSEC|;$$

$$neighbors(G, n) = \{n_1 \in (G.V - n) \ \texttt{s.t.} (n, n_1) \in G.E\}.$$

If $NSEC = \emptyset$, then use Q_i, the pseudo-random number generator, to connect single-node components of the running solution G until $|components(G)| = 1$. In this case of an empty $NSEC$, the generator then finalizes the running solution.

In the converse case of $NSEC \neq \emptyset$, after pairing each node of SEC with one of $NSEC$, the function uses Q_i to choose two arbitrary components $C_0, C_1 \subseteq G.V$, in which G is the running solution. Recall the distance function d_f used by **spine frame**. The **pairing frame** generator template uses a similar function $d_f^{(2)}$.

$$d_f^{(2)}(C) = min_{c \in C}(f_a(c)).$$

For inter-component connection, one variant for **pairing frame** uses $d_f^{(2)}$ to select a node $n_0 = d_f^{(2)}(C_0)$ and another $n_1 = d_f^{(2)}(C_1)$. The new edge (n_0, n_1) merges the components C_0 and C_1 into one. And for intra-component connection, choose an arbitrary component $C \subseteq G.V$ by Q_i. Then use Q_i to choose the two nodes $n_0, n_1 \in C$ such that $n_0 = d_f^{(2)}(C)$ and $n_1 = d_f^{(2)}(C - n_0)$. Unlike in the **spine frame** generator template that terminates with the minimum number of edges required for the solution G to have exactly one component, **pairing frame** terminates when one of its parameters called "connectivity" is satisfied. This metric of "connectivity" is defined in the explanation of **random frame**.

The template **random frame** takes a connectivity argument that determines the number of edges to add to an initial running solution G with vertices $SEC + NSEC$ and a null edgeset. For n nodes, there exists $N_E(n) = (\sum_{0 \leq x \leq n-1} x)$ possible edges. The connectivity argument is a real number $r \in [0., 1.]$ and is the ratio $C_r(G) = |G.E|/N_E(|G.V|)$. For each additional edge to be added to G until $C_r(G) \geq r$, initialized to be a completely disconnected graph with $G.V = SEC + NSEC$ and $G.E = \emptyset$, use a pseudo-random number generator Q_i to select a node $n_0 \in G.V$ such that n_0 does not share an edge with all nodes of $G.V - n_0$ ($degree(n_0) < |G.V|$). Then choose another node $n_1 \in G.V - n_0$ such that $(n_0, n_1) \notin G.E$.

2.9.2 Reasoning Behind Graph Generator Template Design

There are an uncountable number of methodologies to generate simple, undirected graphs under conditions arbitrarily defined. The three generator templates that are **spine frame**, **pairing frame**, and **random frame** each suit different objectives in the context of security. Specific objectives include space efficiency (minimizing edges, shortest distance to SEC node) as well as stochastic decisions for graphs of non-deterministic edgesets.

The first template produces a graph G that is least connected (minimum number of edges for one component). The subgraph of G that contains only the $NSEC$ nodes serve as the "spine" to the graph. And the "branches" connected to the "spine" are the SEC nodes. There are some noteworthy advantages to this configuration.The SEC nodes are maximally distanced from each other. This means that for graph traversal procedures(hopping from node to connected node via edges), there is an tendency for shortest distances between the traversal agent at any node location and a SEC node to stay in a minimal and constrained range. The "spine" also acts as a simple path for the traversing agent to efficiently move from one SEC node to another SEC node. The **pairing frame** tends to produce solutions that minimize SEC nodes being neighbors to each other. The procedure allows for less rigidity than **spine frame** due to the possibility of more edges than the minimum required. And the **random frame** does not consider adding connections based on the sets SEC and $NSEC$, allowing for variabilities in shortest distances between $NSEC$ nodes to SEC nodes.

A third-party agent in *Puissec* is able to utilize crawler-like objects called *Crackling*s [explained in the next chapter],equipped with graph navigation methodologies, to traverse a targeted *SecNet* in search of specific *IsoRing* instances to crack. If a *Crackling* C that is targeting *IsoRing* I is at the same node n in the graph, then one of two events occur depending on the status of n.

- n is in $SEC \rightarrow$ brute-force "guess" attack.
- n is in $NSEC \rightarrow$ interception of I that produces "leaked" information as the resultant.

Crackling-to-*IsoRing* interaction occurs if and only if node "co-location" between C and I occurs.

3 The Cracker

The *Cracker* agent is to "crack" a *SecNet* instance. The highly technical nature of real-world modern crytographic security presents different manifestations of "cracking" processes. Most definitions of computer "cracking", acts committed by computer "crackers", are more universally agreed upon. Cracking is a process, consisting of techniques typically prohibited by administration, to gain unauthorized access to information. Some cracking operations additionally aim for exfiltration of vital information, so that the knowledge of protected information accessed by the third-party agent is used to obtain the electronic evidence (software keys, attribution of digital contents, et cetera) pertinent to the target of cracking. Modern-day cracking places great emphasis on circumvention of defensive measures, vulnerability exploitation, and computational induction.[Kra]

The *Cracker* calculates an order-of-cracking,O, that lists the *IsoRing* instances to be cracked, each element a set of identifiers for co-dependent *IsoRing* instances. As mentioned in the ending paragraphs of the previous chapter, the *Crackling* agent is an object that can traverse the graph of a targeted *SecNet* in search of a specific (targeted) *IsoRing*. A *Cracker* has an arbitrary maximum of a processor slots, each dedicated to a *Crackling* instance. The non-negative integer a is an input argument passed by the user of the *Puissec* program. Obviously, a cannot be a number larger than a threshold pre-determined by the computing hardware used, otherwise the program will crash in cases of maximal usage of the a available slots. For a *SecNet* encapsulating a secret such that the largest required co-dependent set of *Sec* instances to be cracked is of size k, then the condition of $k \leq a$ must hold in order for the *Cracker* to have a non-zero probability of fully cracking the secret (all *IsoRing* instances have been accessed).

The *Cracker* relies on a data container called *BackgroundInfo*, so named because it contains important processes such as hypothesis-based guessing and hypothesis improvement through induction.

3.1 Components of *BackgroundInfo*

Some important variables belonging to this data structure must first be explained.

3.1.1 Better Hypotheses Every Cycle

A hypothesis, in mathematical terms, can be implemented using a generic map with space requirements.

$$H : S_0 \to S_1; S_0, S_1 \texttt{ spaces for input-output map.}$$

Puissec implements a structure called *HypStruct* that acts as a hypothesis for the *Sec* information unit. The *HypStruct* has the following qualities:

- identifier of the targeted *IsoRing* instance I.
- index of the targetted local optimum in dimension d.
- sequence of suspected bounds $\overrightarrow{S}$ that $I[d].V$ might be in,

$$\overrightarrow{S} = \{B : B \texttt{ a matrix of dimension } (d, 2)\}.$$

- vector of suspected probability values $\overrightarrow{P}$ such that $|\overrightarrow{P}| = |\overrightarrow{S}|$. Every i'th element of $\overrightarrow{P}$ corresponds to the i'th element of $\overrightarrow{S}$. **These probability values are used to cross-reference with output values from *Sec*.**
- non-negative integer i, used as one parameter for traversing bounds.

The term "suspect" is used instead of "hypothesis" because the explanation is on the implementation of a "hypothesis" to be used by a third-party agent in its attempts to access the true form of a *Sec*, resulting in circular definition. It carries antagonistic connotation, since there is correlation between the term "suspect" and others including "target","adversary", and "unknown adversary".

This quality suits the demands of the *Cracker* structure, the only agent in the program that uses it, against individually-targetted *Sec* units encapsulated in *IsoRing* structures. *HypStruct* is the primary data unit used in improving guesses for the actual targeted value in arbitrary dimension d.

3.1.2 Leaks Improve Value of Hypotheses

In the context of information space mappings, typically in the format of alphanumeric structures, the term "leak" is meant to denote a transfer of material that is protected and/or prohibitive, with respect to the source, by the source to the prohibited. An implementation of this term is found in *Puissec*. The *Leak* structure is applied onto an *IsoRing*, and its design is specialized for a specific dimension of the multiple *Sec* copies. *Leak* is comprised of two main categories of variables, and those are a memory container M_C that is used to store extracted values from an *IsoRing* and a sequence of instructions $\overrightarrow{F}$, each instruction a function of one of these four.

- L_m: the multiple function. Selects a multiple for a float f that is the value at an arbitrary index i of the actual vector $S.V$. Using an input $r \in [0., 1.]$, calculate all multiples of $S.V[i]$ into a vector $\overrightarrow{M}$ sorted in ascending order. Then choose the j'th index of $\overrightarrow{M}$ equal to $round(r * (|\overrightarrow{M}| - 1))$.

- L_i: the identity function. Outputs the identical value at arbitrary index j of actual vector $S.V$.

- L_b: the sub-bounds function that uses two input arguments $r \in [0., 1.]$ and a bounds B that is the superbound for the local optima in $S.L_O$. Then an application of this function $L_b(S.V)$ produces a sub-bound B_s of length-vector
$$\{(B[i,1] - B[i,0]) * r; 0 \leq i < |S.V|\},$$
such that for every j'th element of B_s,
$$B_s[j,0] \leq S.V[j] \leq B_s[j,1].$$
The actual values of B_s are selected by an external pseudo-random float generator Q_f.

- L_h: hop-specified sub-bounds function. Similar to the function L_b except outputs a sub-bound B_s with an additional hop integer h such that traversal over B_s using the parameter h is guaranteed to produce the actual value. [1]

Every function mentioned in this list operate on single float values f at given indices pertaining to the true form $S.V$. For the sequence $\overrightarrow{F}$, every i'th element has a corresponding ratio value $v_i \in [0., 1.]$ that serves as a multiple in a calculation that yields leak-function outputs from $round(v_i * |S.V|)$ arbitrary elements

[1] Traversal over a bounds is to be fully explained later.

of $S.V$. The resultant is a vector V_l with length equal to $|S.V|$ and elements set to NAN (not a number) if not included in the leak-function outputs.

A *Leak* structure is designed to be a vulnerability associated with a specific *IsoRing* instance at copy d (dimension of the representation). For an *IsoRing* with j *Sec* copies, there are j corresponding *Leak* instances, each with an arbitrary sequence of leak-functions. For usage, suppose that for a *Leak* instance L with size $l = |\overrightarrow{F}|$ functions, an *IsoRing* at representation d (associated dimension of L) is **intercepted** (co-location at an $NSEC$ node) for the i'th time, then the i'th function in $\overrightarrow{F}$ outputs a corresponding leak of information from $I[d]$. If the index i falls out of the length of $\overrightarrow{F}$, then there is no output of leaked information.

The cumulative leaked information for $I[d]$ pertaining to a *Leak* L is stored in a memory container M_c that is a mapping

$$M_c : \texttt{leak-function} \rightarrow \texttt{<sequence of leaked information>}.$$

The elements of the output sequences for the specific leak-function are bounds in the cases of L_b and L_h and as values in $\mathbb{R}$ for the other two leak-functions. The hierarchy, in ascending order of added value to a hypothesis, is $\{L_m?L_b, L_h, L_i\}$, where the question mark separating L_m and L_b signifies the probabilistic nature of their comparative value. There can be $j \geq 0$ instances of the same leak-function L_x in a *Leak* L's variable $\overrightarrow{F}$. In these cases, the L that has been interdicted for $|\overrightarrow{F}|$ times is guaranteed to have j samples of leaked information corresponding to L_x.

3.1.3 Review of the Data Container

The *BackgroundInfo* data container is a structure used by a *Cracker* instance against a specific secret encapsulated by a *SecNet* instance. It contains information such as probability maps (lone,dependent,co-dependent), predicted order-of-operations, a map to leaked information obtained during the *Cracker*'s attempts to access the encapsulated *Sec* instances, and expected probability-value outputs from successful attempts to access. These are the relevant variables for the *BackgroundInfo* structure.

- $L_{pr}^{(h)}$: map for hypothesized lone probability values,

$$\texttt{\textit{Sec} idn.} \rightarrow \texttt{optima dim.} \rightarrow v \in [0., 1.].$$

- M_{dm}: the dependency-chain map that determines the order-of-cracking to apply

```
Sec idn → (optimum index,other Sec identity, other optimum index).
```

- S_{cd}: the set of co-dependent nodes.

- M_{dec}: the hypothesized decision-map

  $$M_{dec} : \texttt{Sec idn.} \rightarrow \texttt{optimum index.}$$

- M_{leak}: a map containing leaked information acquired by a *Cracker*,the possessor of the *BackgroundInfo* instance, during a a session of attempts-to-access all *Sec* instances of the targeted *SecNet*.

  $$M_{leak} : Sec \rightarrow \texttt{optimum dimension} \rightarrow \texttt{<leaked information>}.$$

  The order-of-cracking, as defined previously, can be derived from the maps M_{dm} and S_{cd}.

The *BackgroundInfo* structure is not a standalone operator in *Puissec*. Instead, it is designed strictly as an information container for operation by third-party agents (*Cracker* instances). It contains pre-requisite information for its possessor to use against a targeted *SecNet*. Specifically, the map M_{leak} ideally should not be empty before the *Cracker* begins its attempts against the *SecNet*. The information from M_{leak} is used to reduce n-dimensional search space in attempts-to-access *Sec* instances. The hypothesized decision-map M_{dec} narrows the search space of *Sec* instances' local optima. For the *Cracker*'s issue of probability-value fitting (see the previous section) to verify the values of successful attempts-to-access, the *BackgroundInfo* probability maps $L_{pr}^{(h)}$, M_{dm}, and S_{cd} are used to calculate the expected feedback-response probability value for verification with the output values from the targeted *IsoRing*'s feedback-response loop.

3.2 Implementation of a Brute-Forcer

A fair definition for a brute-force attack is a span of activity in which a third-party agent attempts access into protected information by process of trial-and-error.[For] In contemporary computer networks, with technical infrastructure and protocols for communication, there is a lot of diversification in development and adoption. ***Puissec* does not contain a framework for inputting specific details so that the program can remotely operate against real-world protected computer systems.** Not only is it a simulation of network security, but it also does not allow for any other format of brute-force input into the target *SecNet* instance, other than the two-tuple

$$\texttt{(Sec identity,optimum index).}$$

Contrast this relatively simple format for brute-force attacks with systems such as CAPTCHA and two-step verification specially designed to remove the capability of automated inputs, in these cases of computer security, computerized brute-force attempts. Additionally, real-world encrypted systems typically use characters outside of $\mathbb{R}$ for security keys.

The issues involved in implementing a brute-forcing algorithm for use in "cracking" an encapsulated *Sec* instance do not require focus on design for the input format, the two-tuple previously defined. Instead, there is greater interest in the **traversal pattern** applied onto a suspected sub-space of the possible space of solutions. The term "sub-space" is taken to mean "sub-bounds" of the bounds for the n-dimensional vector that serves as the true form of the targeted *Sec* instance. The term "bounds" is defined in this program as a contiguous space in an arbitrary n dimensions. There are innumerable ways to traverse a bounds. To maintain the simplicity of abstraction, there is one specific traversal pattern used to travel a given input bounds. Every n-dimensional point that the traversal pattern outputs is taken as the vector argument for one guess against an *IsoRing*. Implementation of a traversal pattern for a bounds is in another program called *morebs2* (downloadable through Python Package Index repository) [Mor] The specific structure is called a *ResplattingSearchSpaceIterator* (*RSSI*). It possesses iterative capabilities over an input bounds. There are many features that go outside the scope of basic use (predefined constant inputs for methods). For the intents and purposes of *Puissec*'s usage of *RSSI*, there is a proper subset of variables belonging to an *RSSI* instance to observe.

- $\overrightarrow{B}$, the target bounds matrix.
- h, a value in $\mathbb{N}_{\geq 1}$.
- R, termed "relevance function", used for recognizing each traversed point as relevant (1) or not (0).

For an *RSSI* instance R_S and an initial input bounds $\overrightarrow{B}$, R_S iterates over $\overrightarrow{B}$ by a deterministic procedure. There are no differences in output values from the same inputs over multiple iterations.

1. Set the reference point $\overrightarrow{p}$ equal to $\overrightarrow{B}[:,0]$. This is the first point traveled by R_S.
2. For the i'th row of B, increment the i'th value of $\overrightarrow{p}$ by $h_i = (B[i,1] - B[i,0])/h$,
$$\overrightarrow{p}[i] += h_i.$$
If the updated $\overrightarrow{p}$ has already been travelled, terminate traversal by R_S.

   [1.] If the updated value $p[i]$ equals $B[i,1]$, reset $p[i]$ to the first point of the respective bounds, $B[i,0]$. Then decrement i to
$$(i-1) \mod |B.rows|$$
and repeat the above steps.

   [2.] Otherwise, output the updated $\overrightarrow{p}$ and repeat the loop (start at beginning of step 2).

One of the essential traits of traversal patterns conducted by an *RSSI* instance is its **carryover increment**, described in item 2.1 of the procedure. **This trait means that for a bounds B, the point $B[:,1]$ will not be travelled.**

The reason for the naming of this structure as a "resplatting" search space iterator is because it does not necessarily terminate after iterating over a specific bounds. Its relevance function R is applied to each travelled point from R_S on a reference bounds B_{ref} (initially set to B). After the end of iterating over the sequence of points $\overrightarrow{P}_{ref}$ for B_{ref}, the function R produces a subset of points,

$$\overrightarrow{P}^{(1)}_{ref} \subseteq \overrightarrow{P}_{ref},$$

that are registered as relevant by their output from R. During the iteration of B_{ref}, function R is also used to retrieve relevant sub-bounds of the reference B_{ref} through the registering of relevant points traveled. For a relevant sub-bounds $B_{sb} \subseteq B_{ref}$, all points belonging to

$$\overrightarrow{P}^{(1)}_{ref} \bigcap B_{sb}$$

are registered as relevant by R. There is also the condition that the point traveled before $B_{sb}[:,0]$ and the point traveled after $B_{sb}[:,1]$ are **not** relevant (if the extremum point exists), else B_{sb} has to be extended to include those relevant point/s. Every reference bounds B_{ref} traveled by the R_S produces $x \geq 0$ sub-bounds of B_{ref}, adding these bounds to a cache C_{bounds}. After every traveled bounds, set B_{ref} to the next element b_x of C_{bounds}. Remove b_x from C_{bounds} and proceed with traveling over the updated B_{ref}. Terminate R_S activity for the condition of

$$|C_{bounds}| = 0.$$

There are no constraints for programmer choice of R, other than that it must satisfy the data types of input-output space,

$$R : \mathbb{R}^n \rightarrow \{0,1\};;n \in \mathbb{N}_{\geq 0}.$$

"Splat" is an informal term, defined as impact between a non-fluid and a fluid substance.[Spl] From a perspective, the term is an adequate descriptor of the traversal pattern (fluid) onto a search space (non-fluid), primary function of the *ResplattingSearchSpaceIterator*. The RSSI operates in the manner of a magnifier onto a space to search for points of interest through continuous zooming by the objective function R. Every point traveled by an *RSSI* R_S is a splat from the reference,R_S, onto the aqueous space (a bounds). R_S can re-splat over the same points through traveling over the bounds in the cache C_{bounds}. **There is the necessity of the function R to output accurate boolean values from vector guesses**, the traveled points of the *RSSI* R_S. To elaborate, suppose the two vector sequences, $\overrightarrow{P}_{ref}$ (traveled) and $\overrightarrow{P}^{(1)}_{ref}$ (relevant by R), are equal. But then there would be no reduction in search space, besides from the last point of the reference bounds. The *RSSI* was designed to prevent infinite loops in cases where all points of a reference bounds are marked as relevant. The new sub-bounds B' from the reference bounds B does not include the point $B[:, 1]$. Iterations over B, B' produce the sets of points P_{x_0}, P_{x_1} such that

$$P_{x_0} - (P_{x_1} \bigcap P_{x_0}) \neq \emptyset.$$

If there is no feature of ignoring the last point of a reference bounds, for cases where R produces constant output values for the same input vector, the lack of reduction in search space means that the *RSSI* travels in an infinite loop over the same bound.

3.3 General Procedure of *Cracker*

A *Cracker* instance, itself, does not traverse the targeted *SecNet* instance it is attempting to crack. Given a *BackgroundInfo* instance I_b for a *SecNet* S_{net}, a *Cracker* C_R attempts to crack S_{net} according to the order O derived from M_{dm} and S_{cd}. For every element $S \in O$, the *Cracker* instantiates $c \geq |S|$ *Crackling* instances. At least one *Crackling* instance is assigned to crack each *Sec* instance $S' \in S$. The *Cracker* decides an entry node,belonging to S_{net}, for each $S' \in S$. All the *Crackling* instances are placed onto their chosen nodes of entry. Every *Crackling* has one sole objective.

Definition 3.1 (Crackling Objective (in *Puissec*)). For a crackling C_i targeting a *Sec* S_i encapsulated by an *IsoRing* I_i in a *SecNet* S_{net}, the objective of C_i is to be at the same node as I_i. There is a preference for nodes of the $NSEC$ category than the SEC category due to leaked information by intersection of I_i at an unsecured node.

This definition was intentionally kept simple due to the complexity involved in the design of involved algorithms. For every *Crackling* instance C active at a timestamp t, C travels along the undirected edges of the *SecNet* graph G with the use of a structure called a *TDirector*. The *TDirector* is to guide a *Crackling* instance to the location of the targeted *Sec*, which also employs a separate *Crackling* instance to handle the chasing *Crackling*.

4 Anatomy of *TDirector*

The *TDirector* is a structure responsible for the behavior of an associated agent, an instance of *IsoRing* or *Crackling*. Recall in the previous section that *Crackling*. For the *IsoRing*, the *TDirector* acts according to one of two objectives:

- **avoid target**: prevent any *Crackling* from being on the same node as *IsoRing*.
- **radar null**: stays alert for any pursuing *Crackling*s in timespans where there are no *Crackling* instances spotted yet.

For the *Crackling*, the *TDirector* acts for one of the objectives:

- **search for target**: searches for the target *IsoRing* by hopping to nodes selected by pseudo-random number generator until target *IsoRing* is found.
- **capture target**: the mode set when the target *IsoRing* is found. Calculations are made to devise paths to capture the located *IsoRing*.

The *TDirector* structure contains these variables.

- T: *TDir* instance that stores six variables:

  [-] n_l, node location,

  [-] n_t, target node,

  [-] p_v, vantage point (either *IsoRing* (I) or *Crackling* (C)),

  [-] v, non-negative integer specifying the edge distance travelled per timestamp.

  [-] p_n, path in target graph to travel on.

  [-] p_i, index of location in path that the agent is on.
- t_s: the maximum allowed size of previous *TDir* instances to store in memory.
- L_{td}: log of previous *TDir* instances, with maximum allowed size t_s.
- r: radius, non-negative integer specifying the maximum edge distance,from the reference node (location of agent), that a node is made **visible** to the agent.
- G: the subgraph of the target *SecNet* that centers around n_l, the current location of the agent. The maximum distance of each node in $G.V - n_l$ is r, the radius.

Remark. The term **agent** denotes the *IsoRing* or *Crackling* in possession of the *TDirector*. The converse of **agent** is **target**. A *Crackling* targeting an *IsoRing* is complementary to it, and vice-versa.

There is a term, **graph-in-sight**, that is specially defined in *Puissec* for its two categories of **mobile agents**, *Crackling* and *IsoRing*. The objective of a *TDirector* D differs in the cases of its owner being a *Crackling* or *IsoRing*. At every timestamp (assume all timestamps are set to 1 time unit), an active instance D increments an existing or updated *TDir* instance (its variable T) according to the specified velocity $T.v$ such that the new location is at the $(T.p_i + T.v)$'th index of the path $T.p_n$.

For the *IsoRing* I, the objective function in the mode **avoid target** is

$$\mathbb{O}_{at} = \begin{cases} \max_{p \in PATHS}(s_{path}(T.apply(p, I), T.n_l) & \text{if } |SEC| = 0, \\ \min_{n \in SEC}(s_{path}(n, T.n_l) & \text{otherwise.} \end{cases} \tag{1}$$

Remark. The call *T.apply* changes the location of the agent I to the new location specified by the active path and integer velocity. The objective function attempts to find the best path for the *TDir* $I.D.T$ to travel on. The best path may be equivalent to the one already active on $I.D.T$, in which case the path is continued to be traveled on.

For *IsoRing* I's mode of **radar null**, the objective function is "do nothing",

$$\mathbb{O}_{rn} = \emptyset.$$

Crackling instances operate by different means for their modes **search for target** and **capture target**. The functions involved in calculations for *TDirector* instances belonging to *Crackling*s are generally more complex. Many automata designs were compared with each other, and the general idea is that *TDirector*s associated with *Crackling* instances require more complex functions to pursue or chase the target *IsoRing*. To elaborate, an *IsoRing* by objective function $\mathbb{O}_{rn}$ does not make any decisions when it is in **radar null**. However, for the *Crackling* that does not know the location of the target *IsoRing* I, given its current subgraph G of radius $T.r$ around its own location, resorts to a stochastic search pattern on the target graph until I is found. The search pattern depends on a set S_t of previous target nodes travelled, denoted by variable $T.n_t$. The objective function for moving is the following.

$$\mathbb{O}_{st} = \max_{n \in G.V - S_t} (s_{path}(n, T.n_l)).$$

In the event of tie-breakers for maximum edge-distance in candidates, use a pseudo-random number generator to make a selection. If $G.V - S_t = \emptyset$, then reset S_t to $\emptyset$. The stochastic search procedure sets a new target node $n_i \in G.V$ only when the previous target node $T.n_t$ has been reached by the path p_n.

Once a target *IsoRing* has been located on a *Crackling* c's graph of vision G, the mode switches from **search for target** to **capture target**. Recall that SEC nodes are preferred for an *IsoRing*'s *TDirector*. For *Crackling* instances, there is the converse of $NSEC$ preference for sites of co-location with the target *IsoRing* I, so that interdiction produces leaked information on the *IsoRing* of interest. There is a specific problem that a pursuing *Crackling* has to resolve to successfully accomplish the objective **capture target**, instead of the alternative outcomes of having to continually pursue the target or losing the target (mode gets reset to **search for target**).

There is a specific problem that a pursuing *Crackling* has to resolve to successfully accomplish the objective **capture target**, instead of the alternative outcomes of having to continually pursue the target or losing the target (mode gets reset to search for target). The error is related to the classic **off-by-one** problem prevalent in mathematics [Off]. However, in the specifications of *Puissec*, the concept of off-by-one pertains specifically to the edge-distance between two agents that are not on the same node. With equal velocity (edge-distance per time unit) between the two agents C_i and I_i, there is a strong likelihood that for the location delta between a time unit u and next unit $(u+1)$,

$$s_{path}^{(u)}(loc(C_i), loc(I_i)) = s_{path}^{(u+1)}(loc(C_i), loc(I_i)),$$

on the supposition that C_i and I_i in sight of each other chooses the best paths given their modes of objective, **capture target** and **avoid target** respectively. The challenge can be geometrically represented as a triangle. The three vertices are C_i's own node location, the location of I_i, and the expected location of I_i for the next timestamp (after its travel). There is the attribute of **open information** in *Puissec* simulations. This boolean attribute refers to the information known by any pair of complementary agents (C_i, I_i). If there is no **open information**, then C_i does not know where the visible target I_i's next location is, and vice-versa. But if there is **open information**, then there is a a "toss-up" algorithm used to assign tactical advantage. The "toss-up" goes this typical course:

- Both C_i and I_i know each other's locations and next locations. An agent A, either one of C_i or I_i, determines that there exists a better path P, such that for the updated path p_1 in comparison with the original path p such that $p_1 \neq p$,

  $$s_{path}(T.apply(A, p_1), B)?s_{path}(T.apply(A, p), B).$$

  In the above inequality, agent B is the **complement** to A. According to the objective functions for *Crackling* and *IsoRing* instances, the ? symbol would be $>$ for A as *IsoRing* and $<$ for *Crackling*.

- Then the other agent B, due to open information, is alerted to the fact that agent A changed its path from p to p_1. Agent B then updates its path from p' to p'' in accordance with the inequality expressed in the previous step.

- The agent A is alerted and changes its path in an equivalent way to agent B and the updated path from p to p''.

Remark. In addition to knowing the adversary's updated path, there is also given the knowledge of the velocity to travel on the path for the next timestamp.

Remark. For the remainder of this paper, the term **complement** is taken to mean an adversarial *Crackling* from the vantage point of an *IsoRing*, and the target *IsoRing* from the perspective of a *Crackling*.

A simple resolution for dealing with this infinite loop, described, involves using a pseudo-random integer generator $G_{\mathbb{N}}$ to determine the non-negative integer i of oscillations (each change in agent path counts as one oscillation). The possible range to select the integer is kept small for computing resources, such as $[2, 10]$. And the resolving algorithm starts with an agent selected by $G_{\mathbb{N}}$ to execute the i oscillations in path deltas.

4.1 Analysis Functions Used in Decision Problem of Traversal

Going back to the setting of **no open information**, each traveling agent can predict the path the other will take based on analysis functions alongside pseudo-random generators as selectors. There are two categories of functions provided in *Puissec*, called **target node analysis** and **destination node analysis**. These functions operate differently based on the vantage point of *Crackling* or *IsoRing*.

For **target node analysis**, the function takes as input a node from the agent's graph-in-sight G, and a critical point function f_c (one of minumum,maximum,mean) that accepts a sequence of values in $\mathbb{R}_{\geq 0}$. For every input node $n \in G.V$ as a candidate node,

$$A_t(n) = f_c(\{s_{path}(x, n); x \in l_{comp}\}).$$

In the above equation, l_{comp} is the set of node locations for the complementary agents. The equation is one metric used to gauge the quality of each node, as a destination for the agent with regards to its graph-in-sight. More specifically, the agent knows distances of nodes to the locations of complementary agents.

Another metric is termed **destination node analysis**.

Remark. The naming of the functions may lead to some confusion due to their underlying calculations. Instead of outputting a value in $\mathbb{R}$, A_d produces a map,

```
location of complement → (distance to node)/(mean distance to SEC nodes).
```

The definition of A_d (in mathematical notation) for nodes $n \in G.V$ is

$$A_d(n) = \{(c_i, s_{path}(c_i, n)/mean_{n_2 \in SEC}(s_{path}(n_2, n))); c_i \in l_{comp}\}.$$

Out of the critical point functions used to produce singleton values (elements of $\mathbb{R}$) for outputs from node analysis function A_d, the most common one is min. The reason for this preference is made obvious after the full explanation of how *Crackling* instances pursue a target *IsoRing*.

The two scores are used together to produce a hypothesis for the target's next node of destination. The scores of candidate nodes from node analysis function A_t are used to rank the nodes in ascending order. By the specification of *Puissec*, velocity values are selected in the range [1., 10.]. The issue of selecting a velocity v for a *TDirector*'s next time stamp's travel is a decision problem. In **open information** mode, the velocity is known, so there is a typically smaller set of possible destination nodes for the complement. The set would be all nodes of score from A_t equal to v. In the case of **no open information**, there is a **weak learner** (defined in the next section) used to predict the velocity of the complementary agent/s.

From the vantage point of a *Crackling*, its target *IsoRing* will always attempt to travel to the nearest *SEC* node if *SEC* nodes exist on the graph. Otherwise, it is more difficult to generalize the behavior of the fleeing *IsoRing*. The process for cases of the vantage point being *IsoRing* in a *Puissec* simulation is simpler than that of *Crackling* by default. Since the *IsoRing* instance does not detect any threats to its existence in *SecNet*, it does not move. When its *TDirector* is in the objective mode of **avoid target**, its traversal decisions revolve around the objective function of maximizing distance between it and the *Crackling* pursuer/s.

If a *Crackling* agent C_i were in the mode **capture target**, then there is the probability that the target *IsoRing* will be at least 1 edge-distance from its current location registered by the *TDirector* belonging to C_i at the next timestamp. The *TDirector* predicts a velocity v_i for every complement A_i. Using velocity values v_i for each A_i, a destination node n_i for A_i is calculated, and the agent calculates a path p_i that each A_i takes to reach its destination node. A concept called *path of coincidence* is used by A to calculate the path it will take. For two directed paths $p_x^{(1)}, p_x^{(2)}$, the path of coincidence is a set of nodes $S_n = p_x^{(1)} \bigcap p_x^{(2)}$. The choice of "path" in "path of coincidence" is somewhat of a misnomer, since the set S_n may or may not be ordered. The variable S_n is inputted into a cost function F_{cost}. For simplicity, F_{cost} can be

$$F_{cost}(S_n) = |S_n|.$$

More elaborate candidates for F_{cost} involve weighing the nodes of intersection by their relative ordering in the two directed paths. For example, suppose there are two directed paths p_1, p_2 corresponding to an *IsoRing* and *Crackling*, respectively. Then the F_{cost} function could first reverse the directed path p_2 to p_2'. Then it assigns weights, corresponding to ascending index order, to the path of coincidence S_n. The reversal of p_2 is necessary due to the differing objectives of the *Crackling* (to decrease distance) and the *IsoRing* (to increase distance). The probability of the number of complements being greater than 1 is 0 in the case of the agent A being the *Crackling*.

4.2 The N vs. 1 Capture Problem

As already mentioned, it is possible by the programming of *Puissec* for a *Cracker* C to deploy more than one *Crackling* instance per targeted *IsoRing* I.

The use of multiple *Crackling* instances $\overrightarrow{C}_x$ allow for easier capture of the target in most cases. This is due to "cornering" and "surrounding" formations by $\overrightarrow{C}_x$ with regards to I. Below are two definitions that describe these formations in *Puissec*'s application.

Definition 4.1 (surrounding (in *Puissec*)). A set of *Crackling*s $\overrightarrow{C}_x$ targeting an *IsoRing* I engages in the objective of "surrounding" by navigating in a formation that satisfies some conditions. There is a function G_{corner} that outputs the subgraph of the *SecNet* graph G, by the paths between the node locations of $\overrightarrow{C}_x$.

$$N_x(\overrightarrow{C}_x) = \{PATH(loc(C_i), loc(C_j)).V : C_i, C_j \in \overrightarrow{C}_x\}.$$

Function N_x uses the function $PATH$, which outputs a shortest path between two *Crackling* instances, to accumulate all nodes "encircled" by $\overrightarrow{C}_x$.

One objective is that

$$cap_1 = loc(I) \in N_X(\overrightarrow{C}_x).$$

Over the duration of time units t, there are two variants on minimizing distance of $\overrightarrow{C}_x$ to I:

$$T_1(\overrightarrow{C}_x, I) = \min_{C_i \in \overrightarrow{C}_x} s_{path}(I, C_i),$$

$$T_2(\overrightarrow{C}_x, I) = \min \min_{C_i \in \overrightarrow{C}_x} s_{path}(I, C_i).$$

For time unit t and its next, $(t+1)$, the second objective is

$$cap_2 = T_x(C_x, I)^{(t+1)} < T_x(C_x, I)^{(t)}.$$

Another way of expressing cap_2, using the set p_{dec} (possible paths of travel for C_x to take), is

$$cap_{2'} = \min_{p \in p_{dec}} T_x(C_x, I).$$

Both expressions of cap_2 are to reduce the node size of $N_x(\overrightarrow{C}_x)$ over time units.

Definition 4.2 (cornering (in *Puissec*)). The objective of "cornering" is similar to "surrounding" in the aspect of distance reduction between the set of *Crackling*s $\overrightarrow{C}$ and the targeted *IsoRing* I. However, "cornering" does not use the subgraph function N_x. There is a different focus in terms of paths. Recall that every *Crackling* instance is attached to a *TDirector* that calculates and executes travel directions. A *TDirector* uses its graph-in-sight G' that is a subgraph of some radius r about the center node, the *Crackling* location. Every *TDirector*, through its graph-in-sight, is given the information of what nodes of G' are peripheral nodes, $P' \subseteq G'.V$. The task of $\overrightarrow{C}$ is to pursue I into the "corner" nodes P'. Data structures are used to aid in this decision problem. A log of every traveled node $\overrightarrow{N}$ for each *Crackling*. A task for every *Crackling* $C_i \in C_x$, given its log of traveled nodes is

$$loc^{(t+1)}(C_i) \notin \overrightarrow{N}.$$

Sources

[Gol10] Oded Goldreich. *A Primer on Pseudorandom Generators.* Rehovot,Israel: Weizmann Institute of Science,Department of Computer Science and Applied Mathematics, 2010.

[Lec] URL: `https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture3.pdf`. (accessed: 7.21.2024).

[Off] URL: `http://www.cs.iit.edu/~cs561/cs115/looping/off-by-one.html`. (accessed: 7.28.2024).

[For] *Brute-Force Attack.* URL: `https://www.fortinet.com/resources/cyberglossary/brute-force-attack`. (accessed: 07.06.2024).

[Com] *Computer Security.* URL: `https://www.britannica.com/technology/computer-security`. (accessed: 07.06.2024).

[Csi] *Computer Simulation.* URL: `https://www.britannica.com/technology/computer-simulation`. (accessed: 07.14.2024).

[Kra] *Hacking vs. Cracking.* URL: `https://bluegoatcyber.com/blog/hacking-vs-cracking-clarifying-the-differences-in-cyber-terminology/`. (accessed: 07.14.2024).

[Mor] *morebs2.* URL: `https://pypi.org/project/morebs2/`. (accessed: 7.19.2024).

[Obs] *Obscurantism.* URL: `https://www.oxfordlearnersdictionaries.com/definition/english/obscurantism`. (accessed: 07.09.2024).

[Pri] *Private Methods in Python.* URL: `https://www.geeksforgeeks.org/private-methods-in-python/`. (accessed: 7.19.2024).

[Spl] *splat.* URL: `https://dictionary.cambridge.org/dictionary/english/splat`. (accessed: 7.20.2024).