String-Based Negative Selection inspired by Immunological Strategies applied to Computer Security

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science
of the
University of Aberdeen.

1 4 9 5

Department of Computing Science

Author: Juan-Manuel García del Muro Navarro
MSc. Artificial Intelligence
University of Aberdeen
Student ID: 51771176
Supervisor: Prof. George M. Coghill

Declaration

I declare that this document and the accompanying code has been composed by myself and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Sign: [illegible] Date: 9 August 2018

Juan-Manuel García del Muro Navarro

Acknowledgements

I would like to thank my supervisor, Prof. George M. Coghill for his assistance throughout the project and his flexibility. I thank Biologist Claire Edwards for helping me understand the biological components necessary for the project. I thank my friends for supporting me throughout this journey.

And I thank my family for always supporting me in everything I do. It means the world to me.

Abstract

Basing technologies on nature is not an uncommon occurrence. Like when the Wright brothers first invented the plane, they based their designs on birds. It is no different in the computer science field, where a good amount of the most successful algorithms are based on biology and neuroscience, like Artificial Neural Networks (Also used in Deep Learning) or Genetic Algorithms. It has been proved many times that biological inspired systems can perform extremely well, and that is why we have decided to model an algorithm based on immune systems.

Immune systems are present in most living beings, and they protect them from many threats, including some that are unknown to the system. The way this is done is by T-cells, that are generated randomly and in large numbers, with the aim that every threat to the living being is detected by at least some of these cells. As T-cells are generated randomly, it could occur that one was generated to detected essential parts of the living being as threats. That is why all newborn T-cells have to go through a process of Negative Selection. They are shown self-proteins which belong to the host, and if a T cell detects any of those proteins, it will be destroyed. Thanks to that process, T-cells only detect proteins that are not part of the individual, so they can be dealt with.

This can be potentially very useful when it comes to computing systems and specifically in network security, as it is tricky to defend a system from unknown threats. Our goal in this project is to research the feasibility of various approaches, build an immune-based computer security system and review its performance.

Table of Contents

List of Figures

1 Introduction

In this chapter, we will speak about what inspired us to work on this topic, what immunological systems are and the goals of the project.

1.1 Inspiration

Basing technologies on nature is not an uncommon occurrence. Like when the Wright brothers first invented the plane, they based their designs on birds. It is no different in the computer science field, where a good amount of the most successful algorithms are based on biology and neuroscience, like Artificial Neural Networks (Dasgupta, 1997) (Also used in Deep Learning) or Genetic Algorithms (Gao, Ovaska, & Wang, 2006). It has been proved many times in the past that biological inspired systems can perform extremely well (Dasgupta & Forrest, 1999).

1.2 Immunological systems

Immune systems (Cook, 1989; Dasgupta & Nino, 2000; Elberfeld & Textor, 2009; Gray, 1998; Sha et al., 1988; Starr, C Jameson, & A Hogquist, 2003; Surh & Sprent, 1994) are present in most complex living beings, and they protect them from threats. Those threats sometimes include entities that are previously unknown to the system, and here is where the complexity of the problem relies. The immune system needs to be able to differentiate organisms that are part of the system which the immune system is part of and any other organisms.

This might not seem like an extremely complex task; the system simply needs to be able to identify any component that is part of itself. The problem comes when we delve into the biological part, as we quickly find out that there are an extremely high number of components. And not only that, but most of these components aren't even built with the proteins and tissues that the system is built. An example of these are bacteria, for example, that inhabit almost every complex life form, isn't part of it but becomes absolutely essential for the organism's survival. For that reason, an immunological system needs to not only be able to ignore and tolerate the components of the system itself but a set of external organisms that are beneficial to the body. And such complex task requires a sophisticated solution that scientists have called T-cells.

The first step is to set the boundaries of the system. So, to begin with, we are working with a complex organism, with many different components and processes. There are also threats to the complex organism that have happened to enter its system and have ended up in the bloodstream. As we have stated before, we need a mechanism that detects those anomalies and deals with them, eliminating or neutralizing them. We can define two different processes at this point: The detection of the anomalies and the neutralization.

The way the detection of anomalies is done is by T-cells. These cells are generated randomly and in large numbers, with the aim that every threat to the living being is detected by at least some of these cells. The cells are generated randomly so each one of them detects and tags completely different targets. This process implies that is really frequent that some of the generated T-cells detected essential parts of the system as anomalies or threats. That is why every newborn T-cell has to go through a process of Negative Selection (Blackman, Kappler, & Marrack, 1990; Esponda, Forrest, & Helman, 2004; Liston, Lesage, Wilson, Peltonen, & Goodnow, 2003; Negishi et al., 1995).

The process of Negative Selection happens in isolation and consists of showing the newborn T-cell self-proteins that belong to the host or essential components for its survival, and if the T-cell reacts to these proteins, it will be destroyed. According to this premise, the remaining T-cells that undergo the process successfully without being destroyed are the ones that didn't react to the necessary components for the organism. These same T-cells will react only to what the systems has considered not to be part of the host, what we will call from now on anomalies. These T-cells are ready start its function and they circulate the host's organism. During this process, they encounter different entities. If a certain entity triggers the T-cell meaning that it's considered an anomaly, it will tag it with a chemical beacon and the T-cell will continue circulating the host's system.

On the other hand, the system needs to neutralize the anomalies that have been tagged by T-cells, although once they have been tagged, their neutralization is rather trivial (at least conceptually). White blood cells are very common and can be found in almost every relevant component of the system. They will systematically attack and neutralize any organism that has been tagged with a chemical beacon by a T-cell.

These systems have been improved over millions of years through natural selection, but it's still not perfect. The perfect example is the HIV virus (Aji et al., 2015). In the early stages of the infection, the virus implants its DNA into the T-cells and if successful, it makes it impossible for the immunological system to detect the virus in the blood.

Nevertheless, this system deals with an extremely complex problem in a very efficient way, which is the reason why it has been a source of inspiration for our project.

1.3 Parallelism with Computer Security

Just as biological organisms have to deal with potential threats to their system, computer systems (Balachandran, Dasgupta, Nino, & Garrett, 2007; Bendiab & Kholladi, 2010; Dasgupta, 2006) have to periodically look for malware and neutralize it. As in the case of biology, the host system might have not been in contact with a specific type of malware before, but it still needs to classify (Gonzalez, Dasgupta, & Kozma, 2002) it as a threat. We need a system that is able to react only to entities that are not part of the system.

1.4 Goals

Our goal in this project is to research the feasibility of various immune-inspired approaches, build an immune-based computer security system and review its performance and the potential of this technology.

2 Background

In this chapter we will delve into the background, all the necessary information to fully understand the project. This includes an introduction, the way we will model the information of the system, execution and time complexity, what chunk detectors and contiguous detectors are and how they work, why minimal form is important and the basic mechanics for the project.

2.1 Introduction

In the previous section we spoke about how biological immune systems deal with anomaly detection and neutralization. In this section, we will focus on ways in which that same methodology can be adapted and implemented as a computer security system (Dasgupta & Gonzalez, 2002).

As our organism is now a computer (or an operative system), we will define a set of parameters that will be extracted from the system and we will call it 'self'. Its main function is to represent what normal behavior is for a given system.

The data extracted from the system consists of a list of the processes running on the operative system at the time of execution. It includes the name of the process, path to executable, pid, cpu usage, memory usage and time of execution. This data conforms 'self' (Forrest, Hofmeyr, Somayaji, & Longstaff, 1996) and it's what we used as an input of the algorithm.

In biological immune systems, we found T-cells (Greensmith, Twycross, & Aickelin, 2006) that circulated the host's organism looking for anomalies. The way we have modelled that concept is by defining small data structures called Chunk detectors and Contiguous detectors (Elberfeld & Textor, 2009). Depending on the case, we might want to use one type of detector or the other, or like it often ends up happening, both combined. We will explain them in detail in the next sections.

The next logical step is to define a set of detectors, where none of the detectors match any component of self. We will call these detector sets, and we will refer to the process of generating them given a system and the required parameters will be chunk generation or detector set generation.

The method of negative selection (Ji & Dasgupata, 2004) can technically operate with any type of string-based data, so imputing the data that we have extracted from the system directly into the algorithm would be possible. We chose to use a string-based binary approach mainly for execution and time complexity reasons, that we will describe in the next section.

2.2 Modeling the data through string-based binary

In this section, we will focus on the modeling (Riveiro, Falkman, & Ziemke, 2008) of the data extracted from the system. As we have introduced in the last section, we use the complete list of processes running in the system. The essential components of those processes that we decided to use are the name of the process and its path. Our script receives that data in the form of an alphanumeric string, but as we have already mentioned, the use of an alphanumeric input is not feasible for high sized inputs. It has been proved that binary models perform better and are more scalable than alphanumeric models(Kim & Bentley, 2001)(Kim & Bentley, 2002). We will get into the details in the next section, but for now we will just assume that it's the optimal representation for this problem.

The conversion was rather trivial. We simply converted every alphanumeric digit it's binary equivalent and then appended them to get a binary string, representation of the process. Once we have the data in binary, it needs to undergo a process to transform it into the minimal form. We will explain this process in detail later on in this chapter. This set of binary strings, after being converted into their minimal form represent self, the system itself. As in biology, this represents the components that the detectors shouldn't react to, as they are essential parts of the organism.

2.2 Execution and Time Complexity

As we introduced in the last section, an alphanumeric input would technically be possible. The main problem with this and the reason why it's not a viable solution is because of execution and time complexity (Dasgupta & Attoh-Okine, 1997; Elberfeld & Textor, 2009; Idris, 2011). In all existing negative selection algorithms (Du & Zhang, 2011; Stibor & Timmis, 2007; Stibor, Timmis, & Eckert, 2005), if the representation of the model is based on alphanumeric strings, it suffers from a worst-case exponential size regarding the size of the input. Furthermore, depending on the case, like in systems where the input is of a fixed size or where there are guarantees that the input won't exceed a certain limit, the alphanumeric approach can work. In our case though, the input data is significant, even for the simplest examples and approaches. That is the main reason why we decided to use a binary string-based approach.

The main steps of the algorithm, also the most resource-hungry part of the system are the generation of Chunk-Detectors and Contiguous Detectors, and the module that checks if a certain chunk matches self. In the next sections, we will delve into how the detectors work and the generation process.

2.3 Chunk Detectors

We need to build a set of detectors (Kim, Ong, & Overill, 2003)(Amira Sayed A. Aziz, Azar, Hassanien, & Hanafy, 2014)(A S A Aziz, Salama, Hassanien, & Hanafi, 2012) that will not react to self and ideally, that will react to anything that is not part of self. This type of detectors is the simplest and relatively trivial to generate.

As we introduced in the last chapter, the chunk-detectors, as the self are in form of a binary string. Each chunk-detector is formed by two components: the binary string and an integer. The binary string is the key to check whether the chunk matches a specific component of self or not. The integer is used to position that chunk in the component of self that we are comparing it to. And why would we need to position it? The answer is simple. Chunk-detectors tend to be equal or smaller than the components of self that they are being compared. According to the standard approach, we can only compare two binary strings of the same size. That is solved with the integer, that not only allows to select the part of self that the chunk should be compared to, but also allows to reuse detectors simply by modifying the integer.

The comparison between strings is trivial. Two binary strings match if every digit of each of them is equal. When we apply this comparison to strings in the minimal form, the methodology varies slightly. We will explain it in detail in the last section of this chapter.

The generation of chunk-detectors strictly resembles the generation of T-cells in an organism. A high number of detectors are randomly generated. The next step is to show them the self-set. If they react, they are discarded. If they didn't react, they are added to de detector-set.

In the figure below, we can see an example of this concept. On the left, we can find the self-set. On the right we have included a few chunk detectors of size 3. It is important to remark that this are only some of the non-reactive detectors. The whole detector-set, even for such a small self-set would be significantly bigger. That is the reason why using the minimal form detector-set is mandatory in this approach to make it feasible. We will explain in detail how to generate the minimal form later on in this chapter.

01011	(000,1)	(000,2)	(000,3)
01010	(001,1)	(001,2)	(001,3)
01101	(100,1)	(010,2)	(100,3)
	(101,1)	(011,2)	(110,3)
	(110,1)	(100,2)	(111,3)
	(111,1)	(111,2)	

Figure 1: Example of a self-set and a list of chunk detectors that don't match self

We can see an example that illustrates how chunk detectors work. In the example, the chunk detector is compared to the first part of the self-component. These two match, which means that the detector has detected a component of the self as an anomaly (Dasgupta & Majumdar, 2002; Gomez, Gonzalez, & Dasgupta, 2003; Gonzales & Cannady, 2004). Because of this reason, this detector wouldn't be included in the detector set. We have to keep in mind that we are only interested in keeping those detectors that do not react to components of the self.

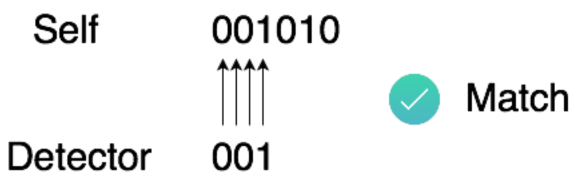


Figure 2: Example of Chunk Detector

We tested different variations of chunk-sets for our approach, and we will get into detail in the implementation chapter.

2.4 Contiguous Detectors

This type of detectors is simpler than chunk-detectors. The basics are the same; contiguous detectors are also generated randomly, and they are compared to the self-set the same way.

The main difference between chunk-detectors and contiguous-detectors is that contiguous detectors, unlike chunk-detectors, are the same size that each component of the self-set. That makes the process of comparison simpler. The integer is not needed either, as the contiguous detector fits the self-component from beginning to end.

Below, we can see an example of a few contiguous detectors for the self in the last example. Again, these detectors aren't in minimal form, and they do not form the full detector set for that specific self.

00000	10100
00100	10110
00110	10111
00111	11000
10000	11001
10001	11110
	11111

Figure 3: Example of a list of contiguous detectors that don't match self (Elberfeld & Textor, 2009)

The following example illustrates how contiguous detectors work. Unlike with chunk detectors, in this case, each detector is compared to a full component of self. That makes the comparison process simpler. In the example, the detector doesn't match self, which means that we are able to store the detector in the detector set, as it will only detect anomalies.

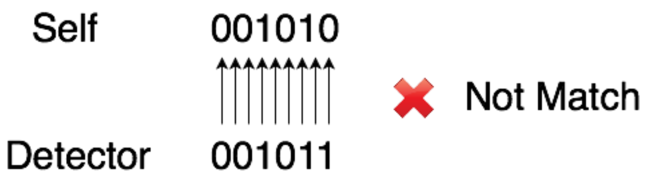


Figure 4: Example of Contiguous Detector

In the next section, we will see the principles of the minimal form, how it works and why it's worth using.

2.5 Minimal Form

In the last section we mentioned that the example wasn't the full detector set. The reason for that is that, even for small inputs, the complete detector set would have to include all the digits that do not match a certain self. For example, if our self was composed by only one binary string of size 3 (extremely rare case, not a real-life example), let's say that S = {111}, then the detector set for contiguous detectors would have to include all of the detectors that don't react to that self (are not equal to self). That would mean that the set would have be: Set = {000, 001, 010, 011, 100, 101, 110}. We can see how that approach although technically possible, is not feasible from an execution, time or space complexity standpoint. Bigger sizes of the components of self would impact the algorithm terribly. For example, in the case of a byte, assuming that there are 8 bits or binary digits, the number of strings stored in the detector set would be of approximately 65 thousand, depending on the size of the input. 8 digits as size of the components of the input is relatively small and not enough to solve most problems. In this project, we have used a list of thousands of strings of 138 bits long. That would mean having a detector set of $3.4 * 10^{41}$ components of size 138 bits long. Clearly not feasible.

The solution to this problem is what the creators of the algorithm named minimal form. This form is a significant simplification of the detector set.

The minimal form set is a set of semi-binary strings. The pool of characters is {0, 1, u}. The reason for that is that our data is still string based, but we need an extra character to represent either 0 or 1, and that is what 'u' represents. That way, the string '00u' includes '001' and '000'. The number of binary strings compiled within a string in minimal form can easily be calculated following the standard the following equation: 2^n where n equals the number of 'u' symbols in the string.

The generation process is much simpler in the binary approach than it is in the alphanumeric one. The first step to generate the minimal form set of detectors is to randomly generate a number of detectors. After, we follow the same process for every single one of those detectors. First, we check if it matches any of the components in self. If it doesn't, we flip the last digit of the string (substitute 0 for 1 and 1 for 0) and we check again if it matches self. If it matches, we leave the digit how it initially was. If it doesn't match either, we substitute that digit for the symbol ‘u’, and we move on to the next digit. We repeat the process with every digit until we reach the end of the string. We can see a very illustrative example below.

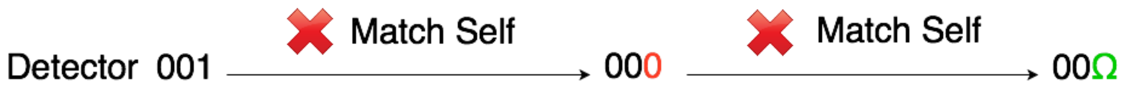
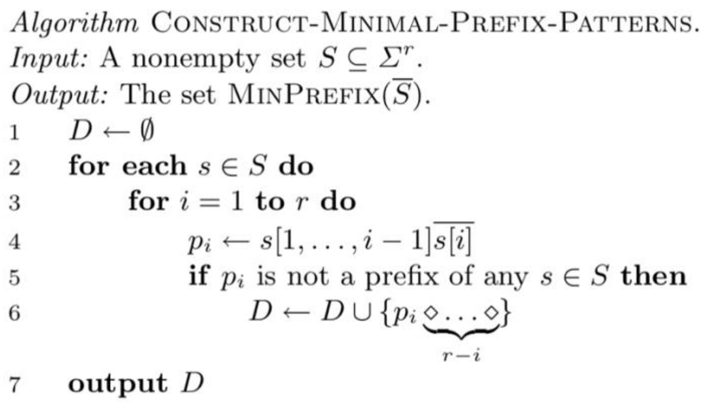


Figure 5: Minimal Form Generation Example

We have also included a simplified version of the generation algorithm (Zeng et al., 2007; Zhengbing, Ji, & Ping, 2008), that can be seen below.



```
Algorithm CONSTRUCT-MINIMAL-PREFIX-PATTERNS.
Input: A nonempty set $S \subseteq \Sigma^r$.
Output: The set MINPREFIX($\overline{S}$).
1   $D \leftarrow \emptyset$
2   for each $s \in S$ do
3       for $i = 1$ to $r$ do
4           $p_i \leftarrow s[1, \ldots, i-1]\overline{s[i]}$
5           if $p_i$ is not a prefix of any $s \in S$ then
6               $D \leftarrow D \cup \{p_i \underbrace{\diamond \ldots \diamond}_{r-i}\}$
7   output $D$
```

Figure 6: Minimal set generation algorithm

When minimal form is used, the size of the detector set can be reduced to the size of the input, which reduces significantly the execution complexity. Binary string and minimal form use reduces the execution complexity from the original worst-case of exponential to a solid polynomial execution complexity (Elberfeld & Textor, 2009; Schapire, 1990), which makes it a feasible solution.

2.6 Mechanics summary

The first step is to define self, which we do by extracting information from the system in which the algorithm is running. With that data, we proceed to generate the detector set in the minimal form. We call this process the training stage, where the algorithm defines what the expected behavior is. Once we have the detector set, our algorithm has finished its training.

At this point, to perform a scan in the system, we simply extract the same information from the system that we used for the training and we run our algorithm, that compares the new information with the detector set and identifies any anomaly that is running in the system.

In the next chapter, we will explain into detail how we modeled the system and we will go over all of its components.

3 Architecture and Implementation

In this section we will focus on the architecture of the project and its implementation. That includes functional and non-functional requirements, technologies used for the implementation, a summary of the approach, a review of the way we have processed and represented information, general review of the system architecture, a meticulous explanation of the file structure and code functions, the steps of the algorithm in detail and the graphical user interface.

3.1 Requirements

3.1.1 Functional Requirements

In this section we will speak about the functional requirements of the system.

- The main functional requirement is that the platform needs to be able to detect anomalies in the system in which it is running.
- The platform needs to be able to extract information from the system in which it is running.
- The platform needs to be able to access the filesystem and store information in files.
- The platform needs to be able to access the filesystem and retrieve information form files.
- The platform needs to be able to extract to process the extracted information.
- The platform needs to be able to process that information and model self.
- The platform needs to be able to compare the model of self with newly extracted information.

- The platform needs to be able to give feedback to the user on the processes that are being performed by the platform, whether the systems was previously trained or not and a list of the detected anomalies in the system.

In the next section, we will speak about the non-functional requirements of the system.

3.1.2 Non-Functional Requirements

In this section we will speak about the non-functional requirements of the system.

- The main and absolutely essential non-functional requirement that we are focusing on is performance. Specifically, we are focusing on designing an algorithm with a worst-case polynomial execution complexity, even though a worst-case logarithmic execution complexity algorithm would also be acceptable. The goal is to achieve an execution complexity below worse-case exponential.
- The system needs to be scalable. We want to be able to not only run it on systems with potentially tens of thousands of processes and components, but we also want to be able to support high numbers of detectors, which improves the detection rate.
- The system needs to be accurate and precise. Specifically, when it comes to false positives, that is crucial to avoid in these types of systems.
- The system needs to be portable. It has to be designed in such a way that it can be run either on Windows, MacOS or Linux.

In the next section, we will speak about the technologies that we have used for the implementation of the system.

3.2 Technologies

The chosen programming language for this project was Python[1]. Its flexibility focuses on prototyping and the fact that it is multiplatform makes it a perfect fit for this project. We used Python 2.7.15 for the development, but it can be run with any version of Python as long as it's within the range 2.7.X. It should also be trivial to run it with the version 3.6 with minima adaptations.

[1] Link to oficial website: https://www.python.org

To build some of the components, a few libraries were absolutely essential. We have the library 'subprocess' to extract the data from the system and store in a data structure. We used the library 'Tkinter' (Shipman, 2013) to build the graphical user interface. The library 'split' helped processing the data. We also used the standard libraries 'random', 'string', 'os' and 'binascii'. We needed to store data in the hard drive, and for that we have used the library 'pickle'.

An important component of the system is the graphical user interface, where we provide the user with useful information, like whether the system has been trained or not, or the progress of the training and anomaly detection search. This interface was integrally built with 'Tkinter' (Shipman, 2013).

As a version control, we have used GitHub. This is the link to the repository, that includes an stable release: https://github.com/astraey/Negative_Selection_Computer_Security (Garcia del Muro Navarro, 2018).

3.3 Summary of the approach

Our approach followed conceptually the general methodology of a negative selection algorithm. The figure below is a diagram that shows the main steps that the algorithm follows.

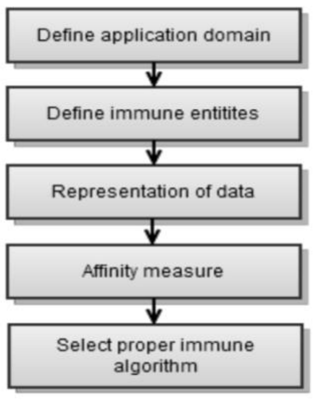


Figure 7: Negative Selection Algorithm (Amira Sayed A. Aziz et al., 2014)

In past sections, we have spoken about negative selection and its mechanics. In this section, we will specify all the steps of our algorithm:

1. Check if there are any trained models stored in the system. If there are, jump to step 4.
2. Extract information from system.
3. Train the model by processing the information and modeling self.
4. Start anomaly detection by extracting information from the system.
5. Process extracted information.
6. Compare processed information to trained model.
7. Analyze results.

These steps are followed every time we run the script. The training is only run once. After that first run, the training and the representation of the system self is stored in a file that will be accessed in posterior executions. If the file is deleted, the system will proceed with the training, starting from step one.

3.4 Information Processing and Representation

The first step after extracting the information about the system is processing it. The data that will conform the system self-set is a list of the processes running in the computer at the time of extraction. That data is extracted by a combination of our script and the python libraries as a data structure, that is then transformed into a list of alphanumeric strings with the names of the processes and their system paths.

That alphanumeric data is then transformed into binary. Every character is replaced by its binary equivalent and then appended together. The last step is to normalize the size of the binary strings. Depending on the length of the path and the name of the process, the length of the binary string can fluctuate significantly. That is why before we can add it to the self-set, we have normalized the binary strings by reducing the size of those that exceed 138 bits by discarding digits starting from the left and augmenting the size of those that don't reach 138 bits by adding 0s at the left of the sting. The last step is to add the binary string set to the self-set. The self-set is stored into a file in the system's filesystem and for later access and that concludes the training.

In the next section, we will delve into the system architecture and we will present a diagram that illustrates the components of the platform.

3.5 System Architecture

The system is formed by a set of modular, well defined components illustrated in the diagram below.

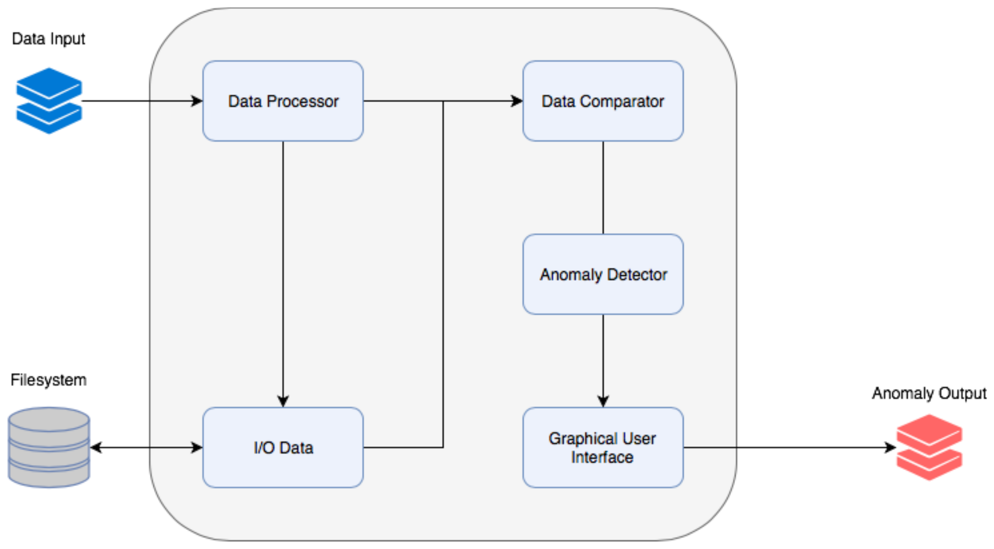


Figure 8: System Architecture Diagram

In our case, the data input is composed by the extracted information from the system. The first component that takes part is the data processor, that refines and formats the data to it can be used by the system. Our platform also needs to be able to access the filesystem of the system in which it's running. That is the main task of the I/O module. It saves and access the trained model so regular anomaly detection runs can be quickly performed. The two next modules are closely related. The data comparator is able to detect matches between the trained model and the newly acquired data. This is done by the anomaly detector module, that outputs a list of the detected anomalies. This data is fed to the graphical user interface, that works as a mediator between the system and the user and shows relevant information about the system and the running processes.

A good way to portrait the components of a project in a technical level is with a class diagram. In this case, we haven't included the methods in each class, as we will explain them in detail in the next section. The class diagram can be seen below.

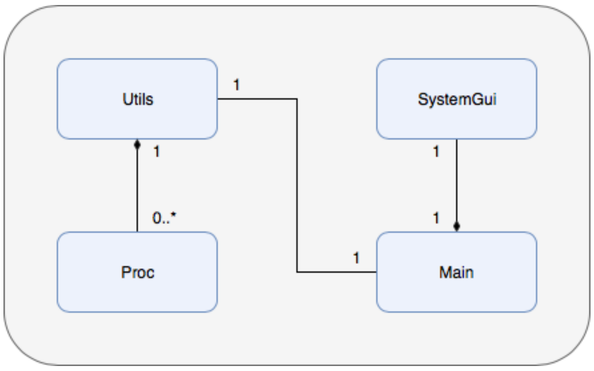


Figure 9: Class Diagram

The diagram is rather simple. We have represented 4 components. Strictly speaking, two of those components aren't classes. 'Utils' is a file with a set of functions, and 'Main' is the main script, but for the proposes of this demonstration, we have added them to the diagram.

In the next section we will delve into the functions and methods that are part of the project.

3.6 Filesystem Structure and Code Functions

3.6.1 Filesystem Structure

The filesystem structure is straightforward, and it includes 6 files.

- process.py: File that includes the class Proc. We designed this class as a data structure to store the extracted information of a process.
- utils.py: This is where we have implemented most of the methods and the functionalities of the project.
- gui.py: Includes all the necessary methods related to the graphical user interface. It's a class that acts as an intermediary between the library Tkinter and other components of our system. A simple graphical user interface controller.
- main.py: The main script. Not much of the logic of the system was implemented in this file. Its main purpose is to work as a central point between all the other components.
- contiguous_list.txt: File where we read and store the trained model of the system. We decided to use a simple text file instead of a formal database for simplicity

purposes. With that said, the next step if we were to escalate the system would be to either include a formal database or to use a specific storage library.

- .gitignore: As we have used the protocol git as a version control, this file is necessary to define what files aren't going to be included in the repository. We decided not to include contiguous_list.txt, as it is specific to the system in which the platform is running and is generated automatically after the training. This way, the paths included in this file are ignored by git.

3.6.2 Functions

The functions and methods that are part of the system has been included in this section. We have classified them according to which class they are part of.

The class 'Proc' is relatively simple, as it was designed as a data structure to hold the information of the extracted processes from the system. It has one method, described below.

- to_str(): Takes no parameters. Returns the user, the pid and the name of the command of a certain process in the form of a string.

The class SystemGUI includes all the necessary methods to interact with the library Tkinter, in charge of the graphical interface.

- addLabel(): Takes no parameters and adds an empty label to the interface.
- changeMessage(message): Takes one parameter, message, which holds the value that we will add to a specific label. The method also renders the label at the bottom of the window.
- startTraining(): Takes no parameters. Modifies a label to show feedback to the user about the status of the system.
- changeStatusMessage(message): Takes the parameter message, which is the value that will be rendered in the label on the interface.
- changeErrorMessage(message, color): Takes two parameters. Message is the parameter that holds the value of the error message that will be displayed on the interface in the error area. The color parameter will define the color of the message on the interface.

The component 'utils' holds the majority of methods and mechanisms of the platform.

- get_proc_list(): Takes no parameters. It uses various libraries to interact with the system, extract a list of the processes running, formats, and returns them.
- stringToBinary(string): Takes the parameter string, which is the string that will be converted to binary and returned as output of the function. The way this function transforms the string into binary is by using the equivalent of each character individually and appends them to generate the result.
- listStringsToBinary(listStrings): Takes a list of strings as a parameter and returns a list of binary strings. The details are trivial, as it uses the function stringToBinary.
- reduceStringList(listStrings): Takes a list of strings as a parameter and returns a reduced version of that list where each component of that list doesn't exceed the value held in the variable maxSelfBinaryStringSize. The algorithm starts discarding digits from the left.
- contiguousGenerator(self, my_gui, root): Takes three parameters. The first one is self, a list of the processes in binary that are representative of what normal behavior. The second parameter is an instance of the SystemGUI class, and directly controls the graphical interface. The third parameter is an extension of the graphical interface, and also selves the purpose of performing tasks on the graphical user interface, like updating the components. This function is crucial in the grand scheme of the system, as is where the logic to generate the detector set was implemented. It takes all of those parameters as an input and after performing negative selection, it discards the useless detectors and returns a list of detectors that corresponds to the detector set.
- generateRandomBinaryString(lenth): Takes one parameter, the length of the randomly generated binary string that this function returns.
- contiguousMatchesSelfSaver(contiguous, self, matchingProcessListIndexes): Takes three parameters. The first parameter contiguous holds the value of an individual contiguous detector. The parameter self represents the extracted list of processes that we want to analyze. This function supports the minimal form, the one where the binary strings can also include the character 'u' that is interpreted as either 0 or 1. The last parameter is the matchingProcessListIndexes, which saves a list of the indexes. The goal of this method is to check if the contiguous detector reacts to any of the binary strings in the list self. If it does, the index of that process is added to the third parameter, matchingProcessListIndexes for later extraction. With that

index number we can identify what process is considered an anomaly by the system and neutralize it.

- contiguousMatchesSelf(contiguous, self): This function is really similar to the previous one. The only differene is that in this case, the function returns true if the contiguous detector matches any of the components of self and it returns false otherwise.
- normaliseLengthStrings(stringList): Takes one parameter, a list of binary strings. It supports minimal form. This function was design to normalize the list of strings. The goal is to return a list of binary strings where all the components have at least a certain length, set by the variable maxSelfBinaryStringSize variable. The procedure is pretty simple. It adds 0s on the left of the string, which doesn't impact the works of the algorithm.
- anomalyDetection(processesList, contiguousList): Takes two parameters. The first parameter is the list of extracted processes that are going to be analyzed. The contiguousList variable holds a list of all the contiguous detectors that have been generated by the system. Here, the comparison algorithm takes part and determines which of those processes are anomalies, based on the contiguous detectors list. It returns the list of anomalies.
- mainScript(my_gui, root): Takes two parameters that, as we have described previously, that control the graphical user interface. Here is where we call most of the functions.

That concludes the list of methods used in our system. The code has been extensively commented, so we recommend taking a look at it to gain a deeper insight on how the functions actually work.

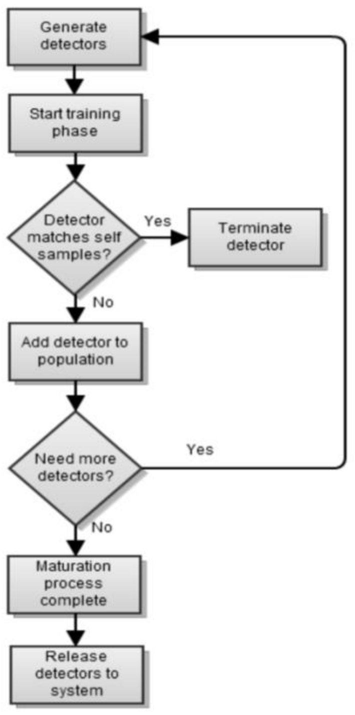
It's also important to state what components from the system architecture diagram correspond to what components, and with that we will finish the section.

The component data processor and I/O Data were implemented in the class 'Utils'. They also use the class 'Proc'. The component Data comparator and Anomaly detector are implemented in both the class 'Utils' and in the main script. Finally, the component Graphical User Interface was integrally implemented in the class 'SystemGUI'.

In the next section, we will speak about the algorithm itself and the steps that it follows.

3.7 The Algorithm in detail

In this section, we will define the main steps that the algorithm (Skiena, 2003) that we have designed follows for the generation of the detector set. We have omitted the anomaly detection steps, as we have explained them in detail in past sections. On the diagram below, we can see the steps that the algorithm follows for the training of the model.



Generate detectors
Start training phase
Detector matches self samples?
Yes
Terminate detector
No
Add detector to population
Need more detectors?
Yes
No
Maturation process complete
Release detectors to system

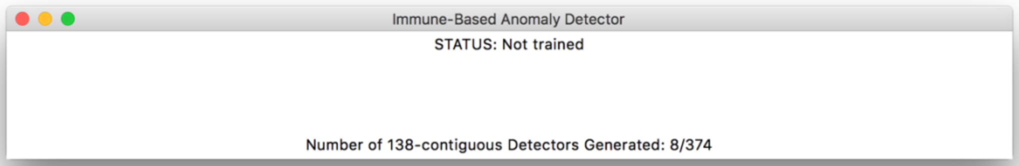
Figure 10: Detectors generation process (Amira Sayed A. Aziz et al., 2014)

The best way to gain an insight on each step is by looking at the code directly and reading the previous section, as it is closely related to this one.

In the next section we will speak about the user interface, why was it necessary and its main components.

3.8 User Interface

We have included a simple graphical user interface to provide the user with information about the system and the processes that it's performing. During the first execution, once the platform detects that the system hasn't been trained yet and it starts the training, the interface shows the training progress by enumerating the total contiguous detectors that are going to be generated and the number that already have been generated. It also shows the length of those detectors. In case of the example, 138 digits. We can see what the interface looks like below.

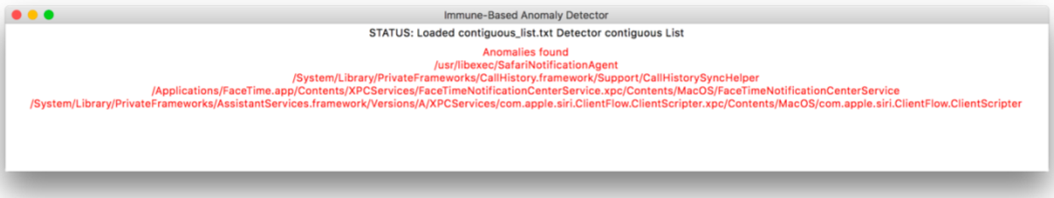


```
Immune-Based Anomaly Detector
STATUS: Not trained

Number of 138-contiguous Detectors Generated: 8/374
```

Figure 11: Graphical Interface during training

Once the training finishes, the system begins with the anomaly detection. The graphical interface changes to show the status of the system and a list of the detected anomalies, in red. In the case of the example, the model hasn't been extracted from the training. It has been loaded from the contiguous_list.txt file, where it was saved from a previous execution.



```
Immune-Based Anomaly Detector
STATUS: Loaded contiguous_list.txt Detector contiguous List
Anomalies found
/usr/libexec/SafariNotificationAgent
/System/Library/PrivateFrameworks/CallHistory.framework/Support/CallHistorySyncHelper
/Applications/FaceTime.app/Contents/XPCServices/FaceTimeNotificationCenterService.xpc/Contents/MacOS/FaceTimeNotificationCenterService
/System/Library/PrivateFrameworks/AssistantServices.framework/Versions/A/XPCServices/com.apple.siri.ClientFlow.ClientScripter.xpc/Contents/MacOS/com.apple.siri.ClientFlow.ClientScripter
```

Figure 12: Graphical Interface during anomaly detection

Those are the two main views of our interface, as the system doesn't require much more.

In the next chapter, we will delve into the system evaluation and testing, and we will speak about its performance.

4 Evaluation and Testing

In this section we will delve into the evaluation and testing of the implementation. That includes the definition of the environment that we have used for testing, the evaluation and testing methodology, a review of the testing and results, and a discussion about the feasibility of the implementation.

4.1 Environment for testing

In this section, we will define the environment in which the system was developed, tested and evaluated.

The computer itself is a MacBook Pro 2016 15" running macOS High Sierra version 10.13.6. The system has a sixth-generation intel core i7 Skylake, 16 Gb of RAM and a Radeon Pro 450 2048 MB.

The script was run with Python 2.7.15, a local installation in the system.

In the next section we will give details of the methodology used and how we tested the performance of the system.

4.2 Methodology

The methodology used was direct and simple. We decided to run the anomaly detection module of the platform on a system where we previously run 5 extra processes that were to be considered anomalies.

The main variable that we can modify in order to adapt the platform to a specific task is the number of detectors that are actually generated. For now, this value is set to the same amount of processes that are extracted from the system. It is worth saying that those detectors are in minimal form, so each of them can be unfolded into a tree structure with millions of leaves that correspond to detectors in non-minimal form. This combines the best of both worlds, as the comparison algorithm is able to access all of that information in the tree indirectly, but we don't have to store those gigantic tree structures in memory.

We have mentioned in the last paragraph that the number of detectors generated can be modified, and that is what we have done when it came to testing. We defined a number of detectors, we ran the detection script 50 times and then we calculated and stored the result, that corresponds with the number of anomalies detected in that environment. That gives us an idea of how well the system performs overall, and when different variables change.

In the next section we will see the results of the testing.

4.3 Testing and Results

The results can be seen in the graph below. The y-axis shows the total number of anomalies detected, out of a total of 5 anomalies running in the system. The x-axis shows the number of detectors that we generated in the model.

It shows how the number of anomalies detected increases linearly, as we increase the number of detectors in the detector set. We can also see how at some point, the algorithm hits plateau as it reaches 4 anomalies detected.

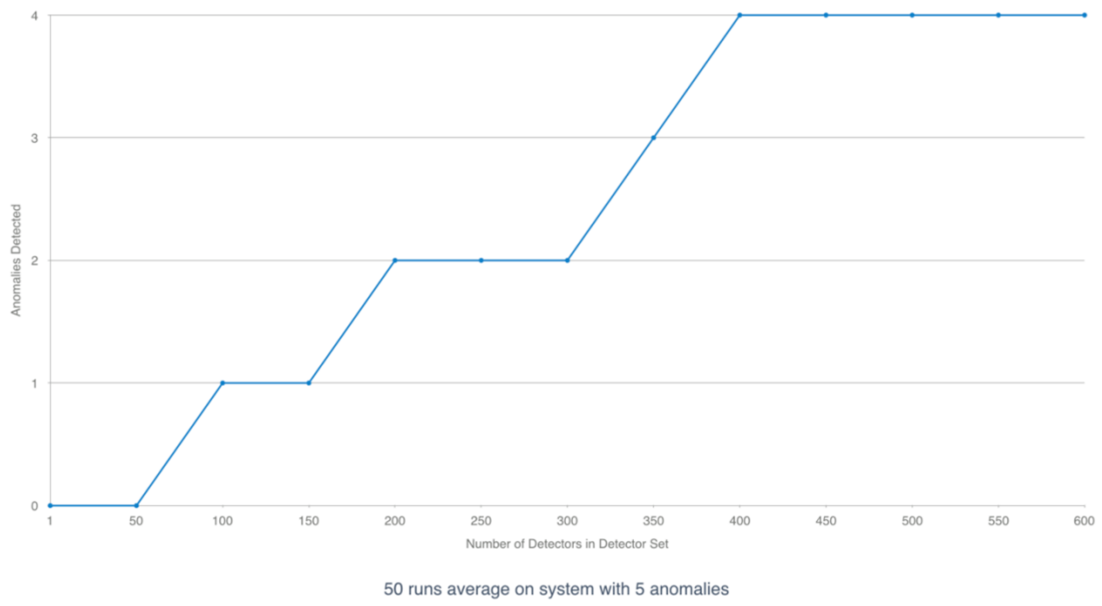


Figure 13: Testing Results

It's also worth mentioning that the system has a very low rate of false positive detections. Even with a detector set of 600, it still wasn't able to detect all the anomalies, but it didn't detect any process that was part of self as an anomaly.

In the next section we will speak about the accomplished functional and non-functional requirements.

4.4 Accomplished Requirements

In the architecture and implementation chapter, we defined a set of functional and non-functional requirements and in this section, we analyze the implementation to determine whether it has accomplished those requirements or not.

4.4.1 Accomplished Functional Requirements

In this section we will review each functional requirement that we stated in the architecture and implementation chapter and we will determine whether they were accomplished or not.

- We stated that the main functional requirement was that the platform needed to be able to detect anomalies in the system in which it is running. This requirement has definitely been met, as our system is able to detect anomalies.

- We stated that the platform needed to be able to extract information from the system in which it is running, and it does, which is the information that is used to model self.

- We stated that the platform needed to be able to access the filesystem and store information in files, and it does.

- We stated that the platform needed to be able to access the filesystem and retrieve information form files, and it does.

- The platform needs to be able to extract to process the extracted information.

- We stated that the platform needed to be able to process that information and model self, and it's able to do that.

- We stated that the platform needs to be able to compare the model of self with newly extracted information, and it does, in order to determine which are anomalies and which aren't.

- We stated that the platform needed to be able to give feedback to the user on the processes that are being performed by the platform, whether the systems was previously trained or not and a list of the detected anomalies in the system, and it does.

So overall, we have accomplished the totality of the functional requirements.

In the next section we will analyze the implementation to determine whether the non-functional requirements have been accomplished.

4.4.1 Accomplished Functional Requirements

In this section we will review each non-functional requirement that we stated in the architecture and implementation chapter and we will determine whether they were accomplished or not.

- The main and absolutely essential non-functional requirement that we are focusing on is performance. Specifically, we are focusing on designing an algorithm with a worst-case polynomial execution complexity, even though a worst-case logarithmic execution complexity algorithm would also be acceptable. The goal is to achieve an execution complexity below worse-case exponential.
- The system needs to be scalable. We want to be able to not only run it on systems with potentially tens of thousands of processes and components, but we also want to be able to support high numbers of detectors, which improves the detection rate.
- The system needs to be accurate and precise. Specifically, when it comes to false positives, that is crucial to avoid in these types of systems.
- The system needs to be portable. It has to be designed in such a way that it can be run either on Windows, MacOS or Linux.

In the next section we will speak about the accomplished requirements.

4.5 Feasibility of the implementation

After the testing, we came to the conclusion that the approach is viable. The execution takes approximately 10 minutes to train the model in the system specified in the past sections. The space complexity was also significantly improved thanks to our binary string-based minimal form.

In the next chapter we will speak about the conclusions of the project, what challenges we ran into, the viability of the proposed approach and potential uses of our platform.

5 Conclusion, discussion and future work

In this chapter we will go over the challenges that we have encountered in this project, as well as the given solutions. We will also discuss the viability of the proposed approach and we will present a potential use for the platform.

5.1 Challenges and solutions

In this section, we will speak about the biggest challenges that we ran into and the way we solved them.

The main challenge, and this is shared by all security systems in computer networks, is that threat come in all shapes and forms, and most of them are unknown to the system. These threats can be direct, like a virus infection, or it can simply be a design flaw that is being exploited. Either way, computer security is a broad, somewhat interdisciplinary field that requires deep knowledge on the subject.

The way we overcame that challenge is by focusing on one particular part of computer security. As we have explained in past chapters, we worked with the running processes in a computer, focusing on detecting anomalies among them.

Another challenge that we run into was that the system not always detected threats. This had many different causes, like the fact that the path or the name of the process of the anomaly had a very similar name to a process that was considered normal. Another cause was that the first version of the implementation suffered from a worst-case exponential execution complexity. That meant that we weren't able to generate enough detectors for the system to detect anomalies.

The solution for the first challenge was to use contiguous detectors, instead of chunk detectors, so even if two processes had the same path, the anomaly wouldn't be considered a normal process. This doesn't fully solve the problem, but it helped reduce the number of anomalies that weren't detected.

The solution to the execution complexity problem was to work extensively in the optimization of the algorithm. We also modified the way we represented data. The algorithm optimization and the use of a binary representation in minimal form helped reduce the execution complexity to a worst-case polynomial complexity, which allowed to generate the number of detectors that we needed to detect most anomalies.

5.2 Conclusion

After a deep technical analysis of the approach and extensive testing of our platform, we have determined that, while it is technically usable, it's not practical to be used as a computer threat detector.

The first issue that made us decide it was the fact that the model needs to be trained, and each training takes about 10 to 20 minutes. Each time that the user install a program, the model needs to be retrained, and that is just not sellable when most antiviruses in the market need to be installed once and don't require any training.
At the same time, we found that our system has a big advantage over any antivirus in the market at the moment; our system is completely standalone independent. It doesn't even require an internet connection, where most antiviruses and security systems do have to be connected to the internet to download the hashes for the latest found threats. Our system solely depends on what processes are running in the computer. But even with this advantage, it would be a hard sell.

The next step after determining that the system wasn't practical at what we originally intended was to focus on the strengths of the system. What our system does best is to detect anomalies based on a previously specified normal behavior. Based on that idea, we came up with a potential application for the system, which we will explain in the next section.

5.3 Potential Application

As we introduced in the last section, what our system does best is to define what standard behaviors for a system are and to detect anomalies.

The potential application would be to use it in servers. Most servers of a certain class are meant to run in a very specific way. Given a server, we could run our system, train it once and run anomaly detections periodically to detect unusual behaviors in the system.

5.4 Future Work

A next step that would improve the system would be to automatize the installation of programs, so every time that the user or administrator of the system installed a program, the model retrained itself to accommodate the new self-components.

Improving the graphical user interface would also make the system more usable and user friendly. We would need to add more buttons and options, so the user had more control over the training of the model, like the number of detectors that are going to be generated, or the frequency in which the system is going to perform tests.

The door is always open to algorithm optimization, and even though we have already optimized it significantly, there's still room for improvement.

Lastly, it would be a good idea to include a verification module. This module would have access to the internet and it would be able to verify program certificates and signatures to determine if it's safe.

It's worth mentioning that we have worked on the project in a public repository and that anyone can participate, so this section should serve as a guide if anybody was interested in continuing working on the project.

6 References

Aji, M. M., Adamu, A. M., Borkoma, M. B., Gharge, S., Menon, G. S., Issn, J., … Branch, I. (2015). Journal of AIDS and HIV Research (JAHR). *African Journal of Environmental Science and Technology*, *7*(4), 944–948. https://doi.org/10.5897/AJEST2013.1532

Aziz, A. S. A., Azar, A. T., Hassanien, A. E., & Hanafy, S. E.-O. (2014). Negative Selection Approach Application in Network Intrusion Detection Systems, (Ci). Retrieved from http://arxiv.org/abs/1403.2716

Aziz, A. S. A., Salama, M., Hassanien, A. ella, & Hanafi, S. E.-O. (2012). Detectors generation using genetic algorithm for a negative selection inspired anomaly network intrusion detection system. In *2012 Federated Conference on Computer Science and Information Systems (FedCSIS)* (pp. 597–602).

Balachandran, S., Dasgupta, D., Nino, F., & Garrett, D. (2007). A Framework for Evolving Multi-Shaped Detectors in Negative Selection. In *2007 IEEE Symposium on Foundations of Computational Intelligence* (pp. 401–408). https://doi.org/10.1109/FOCI.2007.371503

Bendiab, E., & Kholladi, M. K. (2010). The Negative Selection Algorithm: a Supervised Learning Approach for Skin Detection and Classification. *IJCSNS International Journal of Computer Science and Network Security*, *10*(11), 86–92. Retrieved from http://paper.ijcsns.org/07_book/201011/20101114.pdf

Blackman, M., Kappler, J., & Marrack, P. (1990). The role of the T cell receptor in positive and negative selection of developing T cells. *Science*, *248*(4961), 1335 LP-1341. Retrieved from http://science.sciencemag.org/content/248/4961/1335.abstract

Cook, M. (1989). Receptor editing (and the evolution of sex). *Biotechnology*, (1), 55–56.

Dasgupta, D. (1997). Artificial neural networks and artificial immune systems: similarities and differences. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation* (Vol. 1, pp. 873–878 vol.1). https://doi.org/10.1109/ICSMC.1997.626212

Dasgupta, D. (2006). Advances in artificial immune systems. *IEEE Computational Intelligence Magazine*, *1*(4), 40–49. https://doi.org/10.1109/MCI.2006.329705

Dasgupta, D., & Attoh-Okine, N. (1997). Immunity-based systems: a survey. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation* (Vol. 1, pp. 369–374 vol.1). https://doi.org/10.1109/ICSMC.1997.625778

Dasgupta, D., & Forrest, S. (1999). Artificial immune systems in industrial applications. In *Proceedings of the Second International Conference on Intelligent Processing and Manufacturing of Materials. IPMM'99 (Cat. No.99EX296)* (Vol. 1, pp. 257–267 vol.1).

https://doi.org/10.1109/IPMM.1999.792486

Dasgupta, D., & Gonzalez, F. (2002). An immunity-based technique to characterize intrusions in computer networks. *IEEE Transactions on Evolutionary Computation*, *6*(3), 281–291. https://doi.org/10.1109/TEVC.2002.1011541

Dasgupta, D., & Majumdar, N. S. (2002). Anomaly detection in multidimensional data using negative selection algorithm. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)* (Vol. 2, pp. 1039–1044 vol.2). https://doi.org/10.1109/CEC.2002.1004386

Dasgupta, D., & Nino, F. (2000). A comparison of negative and positive selection algorithms in novel pattern detection. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. "cybernetics evolving to systems, humans, organizations, and their complex interactions" (cat. no.0* (Vol. 1, pp. 125–130 vol.1). https://doi.org/10.1109/ICSMC.2000.884976

Du, B., & Zhang, L. (2011). Random-Selection-Based Anomaly Detector for Hyperspectral Imagery. *IEEE Transactions on Geoscience and Remote Sensing*, *49*(5), 1578–1589. https://doi.org/10.1109/TGRS.2010.2081677

Elberfeld, M., & Textor, J. (2009). Efficient algorithms for string-based negative selection. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *5666 LNCS*, 109–121. https://doi.org/10.1007/978-3-642-03246-2_14

Esponda, F., Forrest, S., & Helman, P. (2004). A formal framework for positive and negative detection schemes. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *34*(1), 357–373. https://doi.org/10.1109/TSMCB.2003.817026

Forrest, S., Hofmeyr, S. A., Somayaji, A., & Longstaff, T. A. (1996). A sense of self for Unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy* (pp. 120–128). https://doi.org/10.1109/SECPRI.1996.502675

Gao, X. Z., Ovaska, S. J., & Wang, X. (2006). Genetic Algorithms-based Detector Generation in Negative Selection Algorithm. In *2006 IEEE Mountain Workshop on Adaptive and Learning Systems* (pp. 133–137). https://doi.org/10.1109/SMCALS.2006.250704

Garcia del Muro Navarro, J.-M. (2018). Negative Selection Computer Security. *Github*. Retrieved from https://github.com/astraey/Negative_Selection_Computer_Security

Gomez, J., Gonzalez, F., & Dasgupta, D. (2003). An immuno-fuzzy approach to anomaly detection. In *The 12th IEEE International Conference on Fuzzy Systems, 2003. FUZZ '03.*

(Vol. 2, pp. 1219–1224 vol.2). https://doi.org/10.1109/FUZZ.2003.1206605

Gonzales, L. J., & Cannady, J. (2004). A self-adaptive negative selection approach for anomaly detection. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)* (Vol. 2, p. 1561–1568 Vol.2). https://doi.org/10.1109/CEC.2004.1331082

Gonzalez, F., Dasgupta, D., & Kozma, R. (2002). Combining negative selection and classification techniques for anomaly detection. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)* (Vol. 1, pp. 705–710 vol.1). https://doi.org/10.1109/CEC.2002.1007012

Gray, A. J. T. G. D. (1998). Receptor editing during affinity maturation. *Immunology Today*, (4), 196.

Greensmith, J., Twycross, J., & Aickelin, U. (2006). Dendritic Cells for Anomaly Detection. In *2006 IEEE International Conference on Evolutionary Computation* (pp. 664–671). https://doi.org/10.1109/CEC.2006.1688374

Idris, I. (2011). E-mail Spam Classification With Artificial Neural Network and Negative Selection Algorithm. *International Journal of Computer Science & Communication Networks*, *1*(3), 227–231.

Ji, Z., & Dasgupata, D. (2004). Augmented negative selection algorithm with variable-coverage detectors. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)* (Vol. 1, p. 1081–1088 Vol.1). https://doi.org/10.1109/CEC.2004.1330982

Kim, J., & Bentley, P. J. (2001). Towards an artificial immune system for network intrusion detection: an investigation of clonal selection with a negative selection operator. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)* (Vol. 2, pp. 1244–1252 vol. 2). https://doi.org/10.1109/CEC.2001.934333

Kim, J., & Bentley, P. J. (2002). Towards an artificial immune system for network intrusion detection: an investigation of dynamic clonal selection. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)* (Vol. 2, pp. 1015–1020 vol.2). https://doi.org/10.1109/CEC.2002.1004382

Kim, J., Ong, A., & Overill, R. E. (2003). Design of an artificial immune system as a novel anomaly detector for combating financial fraud in the retail sector. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.* (Vol. 1, p. 405–412 Vol.1). https://doi.org/10.1109/CEC.2003.1299604

Liston, A., Lesage, S., Wilson, J., Peltonen, L., & Goodnow, C. C. (2003). Aire regulates negative selection of organ-specific T cells. *Nature Immunology*, *4*, 350. Retrieved from http://dx.doi.org/10.1038/ni906

Negishi, I., Motoyama, N., Nakayama, K., Nakayama, K., Senju, S., Hatakeyama, S., … Loh, D. Y. (1995). Essential role for ZAP-70 in both positive and negative selection of thymocytes. *Nature*, *376*, 435. Retrieved from http://dx.doi.org/10.1038/376435a0

Riveiro, M., Falkman, G., & Ziemke, T. (2008). Improving maritime anomaly detection and situation awareness through interactive visualization. In *2008 11th International Conference on Information Fusion* (pp. 1–8).

Schapire, R. E. (1990). The Emerging Theory of Average Case Complexity. *Techreport*, *219*(LCS/TM-431), 193–219. Retrieved from papers://5e6aef66-ac63-43ea-b80a-d37150576ab8/Paper/p115

Sha, W. C., Nelson, C. A., Newberry, R. D., Kranz, D. M., Russell, J. H., & Loh, D. Y. (1988). Positive and negative selection of an antigen receptor on T cells in transgenic mice. *Nature*, *336*, 73. Retrieved from http://dx.doi.org/10.1038/336073a0

Shipman, J. W. (2013). Tkinter 8.5 reference: a GUI for Python. *Computer*, 1–118. Retrieved from tcc-doc@nmt.edu

Skiena, S. S. (2003). *The Algorithm Design Manual.*

Starr, T. K., C Jameson, S., & A Hogquist, K. (2003). Positive and Negative Selection of T-cells. *Annual Review of Immunology*, *21*(1), 139–176. https://doi.org/10.1146/annurev.immunol.21.120601.141107

Stibor, T., & Timmis, J. (2007). Comments on real-valued negative selection vs. real-valued positive selection and one-class SVM. In *2007 IEEE Congress on Evolutionary Computation* (pp. 3727–3734). https://doi.org/10.1109/CEC.2007.4424956

Stibor, T., Timmis, J., & Eckert, C. (2005). On the appropriateness of negative selection defined over Hamming shape-space as a network intrusion detection system. In *2005 IEEE Congress on Evolutionary Computation* (Vol. 2, p. 995–1002 Vol. 2). https://doi.org/10.1109/CEC.2005.1554799

Surh, C. D., & Sprent, J. (1994). T-cell apoptosis detected in situ during positive and negative selection in the thymus. *Nature*, *372*, 100. Retrieved from http://dx.doi.org/10.1038/372100a0

Zeng, J., Li, T., Liu, X., Liu, C., Peng, L., & Sun, F. (2007). A Feedback Negative Selection Algorithm to Anomaly Detection. In *Third International Conference on Natural Computation (ICNC 2007)* (Vol. 3, pp. 604–608). https://doi.org/10.1109/ICNC.2007.28

Zhengbing, H., Ji, Z., & Ping, M. (2008). A Novel Anomaly Detection Algorithm Based on Real-Valued Negative Selection System. In *First International Workshop on Knowledge Discovery and Data Mining (WKDD 2008)* (pp. 499–502). https://doi.org/10.1109/WKDD.2008.110

7 Appendix A – User Manual

In this section, we will delve into the platform itself, and how to use it.

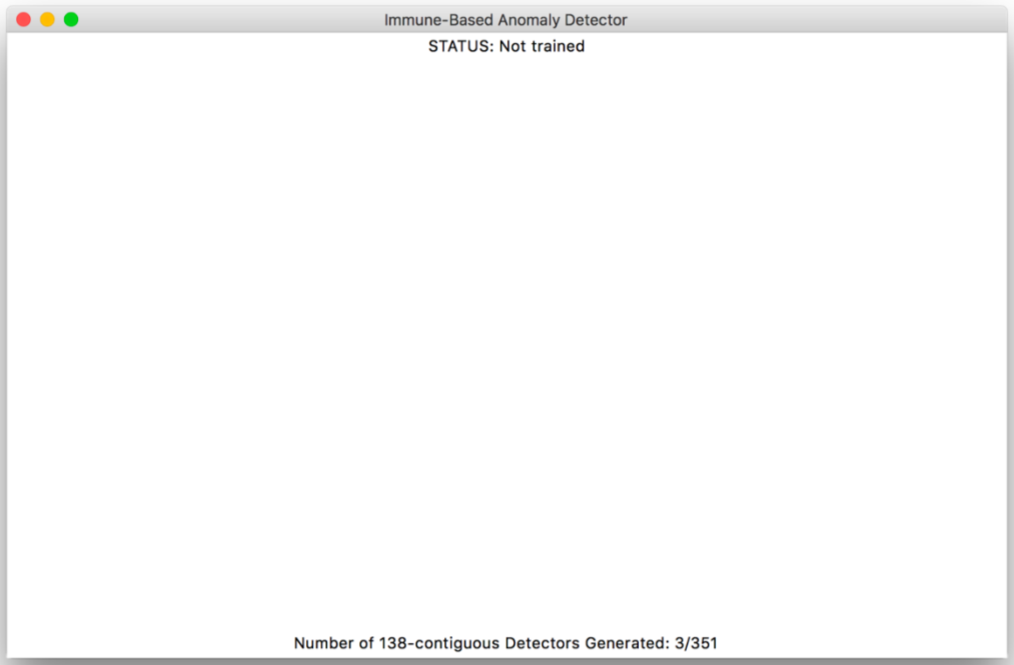
There are two main functions: training and anomaly detection. When you first execute the script, it will look for the file contiguous_list.txt. If such file doesn't exist, the platform will execute the training. If it does detect it, it will retrieve the data that contains the model and jump directly to the anomaly detection.

To run the program, we simply have to run the file main.py. Details on how to do that will be explained in the next section.

In the next sections, we will explain in detail how those two stages work from the user prospective.

7.1 Training

In the training stage, a window opens like the one that can be seen below.



STATUS: Not trained

Number of 138-contiguous Detectors Generated: 3/351

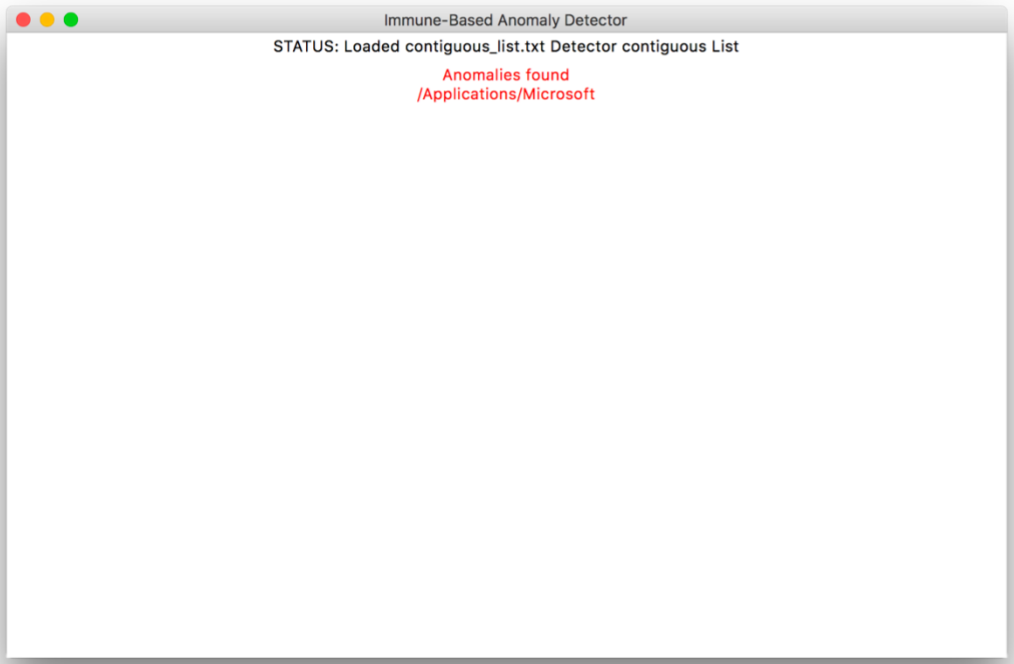
The information shown is self-explanatory. At the top of the windows there is a label that indicates whether the system has been previously trained or not. If the user sees this view, it means that the platform is being trained in the background.

At the bottom of the window we can find a label that indicates the process of the training. The first value indicates the generated detectors and the second value indicates the total number of detectors that are going to be generated. We can also see that in this example, in the bottom label it indicates that the size of the detectors is 138 bits. The standard way of indicating the size of the detector is in the following format: (size value in bits)-contiguous/chunk Detectors.

When the training finishes, the system stores the model in the file contiguous_list.txt and runs an anomaly scan.

7.2 Anomaly Detection

In the anomaly detection stage, the window below appears.


Immune-Based Anomaly Detector

STATUS: Loaded contiguous_list.txt Detector contiguous List

Anomalies found
/Applications/Microsoft

In the case of the example on the window above, the system has loaded the model from the cile contiguous_list.txt. A label that shows that information can be seen at the top of the window. In a label directly below, we can see a list of the detected anomalies. In the example, we can see that the process /Applicatoin/Microsoft has been detected as an anomaly. The reason for that is that that process wasn't running when the model was trained, and it's detecting it as unusual behavior.

In the next Appendix we will speak about the installation and the maintenance of the platform.

8 Appendix – B Installation and Maintenance Manual

In this section we will speak about the installation and maintenance of the system.

8.1 Installation and requirements

The programming language used was Python[2], particularly the version 2.7.15. Any 2.7.X version will work and with very little modification, it can be run with Python 3.6.X.

The system data extraction module was designed and tested on a Mac, but it will work for any Unix-like operative system with a bash terminal.

To run the project, we simply have to execute the following command:

```
python main.py
```

8.2 Dependencies

There're a few libraries that are essential to run the project. They can be installed with the Python package manager pip, simply by executing the following command:

```
pip install [name of the library]
```

We have included a list of the necessary libraries. Keep in mind that the command is case sensitive.

- subprocess
- re
- process
- random
- string
- os.path
- binascii
- pickle
- Tkinter

[2] Link to oficial website: https://www.python.org

The library Tkinter can't be installed with the package manager pip and needs to be a part of the Python installation.

8.3 File structure

When we extract the code folder from the zip, we will see 4 files:

- main.py
- process.py
- utils.py
- gui.py

The system will automatically create a fifth file to store the trained model:

- contiguous_list.txt

In order to retrain the system, the file contiguous_list.txt has to be deleted. If the system is not able to find it, it will automatically start the training process.

8.4 Crucial named constants

The most important constant is maxSelfBinaryStringSize, and it's defined in the utils.py file, at the top. This constant sets the size of the components of both the self and the detector set. For the proposes of this demonstration, the value was set to 138. The modification of such value will impact the system significantly. The smaller, the less the system will take to train the model but the less anomalies it will detect.

8.5 Modifying and extending the program

In this section we will focus on the modifications and extensions that we suggested in the future work chapter.

The first modification, making the system automatically retrain itself when a new program is installed would be included in the main loop, in main.py. We would simply need a trigger that is set every time we install a new program. That trigger would call the function in charge of training the model.

To include a verification model, we would need to add a new class that was in charge of that. To include that class's functionality in the system, we would simply have to include the method calls for the new class in some of the methods that perform the anomaly detection in the component utils, in the file utils.py. As an idea, we could only consider a process an anomaly if both the current detection module and the newly added verification module consider it an anomaly.

The graphical user interface can easily be modified directly modifying the class SystemGUI. To include triggers and reactions in the negative selection algorithm, we can simply add them in the utils.py file, as we have access to the full implementation of the SystemGUI class from that component as well.

The logic of the system was included in the component utils, and if we were to optimize the algorithm, that is where we should do it. Every function has been assigned with a very specific task, so the improvement of the mechanics of the algorithm should be a relatively simple task.

It's worth saying that from the first release of the first classes of the project, everything has been online in an open repository. The full history of commits and releases can be found in the following link: https://github.com/astraey/Negative_Selection_Computer_Security.

8.6 Trace bugs in execution

There is not a specific module in the system to trace bugs, but throughout the development of the project, we have used labels and prints to show key data and values of variables that can be causing bugs. In the latest release, such code has been deleted but it can be accessed downloading the code from a previous commit.